SPARQ: Skew-Aware PIM Accelerator for Relational Join and Select Queries

Abstract

In-memory databases are widely used for large-scale analytics, but their performance is increasingly limited by the gap between CPU speed and DRAM access cost. In these systems, joins are a major bottleneck because they combine rows from different tables and often require many irregular hash-table lookups over large datasets. As a result, join performance is often constrained by DRAM bandwidth and access latency. Processing-in-Memory (PIM) can reduce this cost, but prior PIM join designs still face three limitations: they often reuse CPU-style algorithms, do not fully exploit DRAM-internal parallelism, and suffer from extra inter-chip communication and skewed key distributions.

We propose SPARQ, a DRAM-based PIM design for hash joins and related SELECT operations. SPARQ builds on three main ideas. First, it adds subarray-level comparators, allowing many entries within an activated DRAM row to be searched in parallel. Second, it stores only unique key–value pairs in PIM and keeps duplicate tuple identifiers in a separate duplication-linked list. This deduplication, together with a co-designed row-resident layout, maps each hash bucket to a single DRAM row, replacing sequential chain traversal with bounded in-row lookup. Third, SPARQ adds a hardware Rank-Level Unit (RLU) that exploits DIMM-local placement of probe keys and PIM-resident hash buckets to pipeline key fetch, lookup, and result return within the memory rank, reducing CPU involvement. Together, these techniques exploit fine-grained DRAM parallelism, reduce host-side coordination, and make join latency predictable under skew and duplication.

We integrate SPARQ with DuckDB by offloading JOIN operators to PIM while keeping the remaining query stages on the CPU. Across Star Schema Benchmark queries, SPARQ achieves 12× average speedup for join processing and 5.7× end-to-end query speedup over CPU-only DuckDB, with about 7% data overhead and 2.1% area overhead per PIM-enabled DRAM chip.

1 Introduction

The widening gap between CPU capability and DRAM performance, commonly known as the memory wall [87], significantly limits modern data-intensive applications. This challenge is exacerbated by the difficulty of scaling DRAM bandwidth [19], which increases the cost of data transfer between the CPU and memory.

Modern **in-memory database systems** underpin applications in business intelligence, e-commerce, healthcare, and finance [30, 67]. Among their core operations, joins are particularly critical because they combine rows from multiple tables or relations and involve intensive, irregular memory accesses over large datasets [67]. As a result, join performance is closely tied to memory efficiency and plays a major role in overall query latency.

As Figure 1(A) shows, `JOIN` accounts for a substantial fraction of query latency across individual SSB queries [74], while `SELECT` is the most frequently used operator in real-world workloads [20].

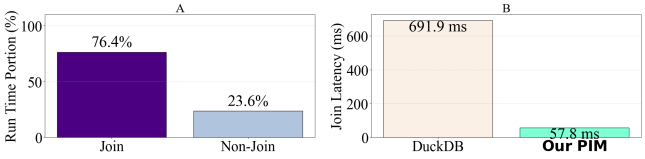


Figure 1: A: Average join time portion in total query time across SSB queries [68], using DuckDB [82] on CPU. B: Average join runtime across SSB individual queries using DuckDB on CPU vs. DuckDB with SPARQ (setup in Sec. 5).

Join and select on CPUs face three core **challenges**:

- **Memory-bound computation**: Memory access bottlenecks account for over 80% of join query latency [17]. Frequent random memory accesses to locate hash buckets, followed by variable-length bucket traversal, increase cache and Translation Lookaside Buffer (TLB) misses, stalling CPU pipelines [16, 18, 45, 76].
- **Data skew**: Skewed key distributions and duplicates degrade join performance, with hash joins being particularly susceptible [14, 27].
- **Low parallel efficiency**: Parallelizing joins adds synchronization, partitioning, and Non-Uniform Memory Access (NUMA) overheads [79], yielding limited and possibly negative speedups [10, 48, 55] and creating the potential for load imbalance that is further exacerbated by skew [14].

Figure 2 shows the Intel Advisor [21] roofline model for DuckDB's join (setup described in Section 5). The x-axis is arithmetic intensity, and the y-axis is achieved performance. The sloped roofs show the peak performance attainable under bandwidth limits at different memory levels, while the flat roof shows peak CPU compute performance. DuckDB's join lies at low arithmetic intensity and far below the bandwidth roofs, indicating that performance is limited by irregular hash-table lookup, cache/TLB misses, and stalls rather than compute capability.

To mitigate the memory-wall bottleneck in join processing and improve overall Database Management System (DBMS) performance, prior work has explored hardware and near-data accelerators [3, 25, 26, 50, 53, 55, 60], such as DRAM-PIM that integrates computation into the memory subsystem. However, PIM operates under tight constraints on compute capability and data movement within and across DRAM devices, and therefore differs fundamentally from CPU execution [8]. Despite this mismatch, much of DRAM-PIM join research adapts CPU-based algorithms [50, 53, 60]. Such algorithms deliver **limited benefit in PIM settings** because they do not fully exploit DRAM's fine-grained internal parallelism and can incur **excessive cross-chip data transfers** [39, 53]. **GPUs** face similar challenges from host–device transfers and skew [14, 78] (Sec. 5.2.3). Consequently, without a skew-resilient, DRAM-parallel, communication-efficient design, data movement and synchronization overheads dominate end-to-end latency.

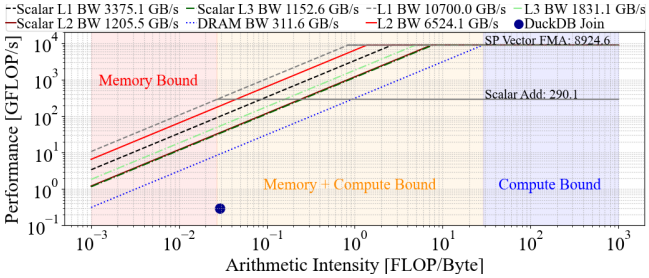


Figure 2: SSB SF100 DuckDB join roofline (setup in Sec. 5).

Goal. We design a scalable, skew-robust PIM join accelerator that exploits fine-grained DRAM parallelism, delivers predictable latency, and minimizes cross-chip and host overheads.

Approach. This paper presents SPARQ (**S**kew-aware **P**IM **A**ccelerator for **R**elational JOIN and SELECT **Q**ueries), a DRAM-PIM design that accelerates hash joins (a widely used PIM-friendly primitive for large in-memory equi-joins [11, 53, 74]) and their associated select operations. SPARQ targets LRDIMM-PIM, a scalable and cost-effective platform that avoids the thermal and scalability challenges of alternative PIM technologies [7].

To exploit parallelism and minimize cross-chip communication, SPARQ stores the hash-join hash table in the PIM-enabled chip of DRAM ranks and performs in-place lookups using subarray-level comparators. Mapping each bucket to one subarray row enables parallel comparison within the bucket and thereby constant-time match detection, avoiding sequential traversal and CPU stalls.

To support a simple hardware-friendly hash function while bounding bucket size to one row, SPARQ maintains a duplication-linked list in off-PIM memory and stores only unique key-value pairs in PIM. This preserves predictable probe latency under skew and avoids frequent rehashing; lightweight encoding further reduces PIM-resident data size and aligns the layout with DRAM's bank/column structure for efficient access.

Each PIM-enabled rank integrates an RLU that operates on a compressed copy of the join-key column (distributed across non-PIM-enabled chips in PIM-enabled ranks and locally traversed). The RLU pipelines key traversal with subarray comparisons, removing CPU involvement and reducing cross-chip coordination and synchronization overhead.

SPARQ offers three **key contributions**:

- **Subarray-level PIM parallel search engine**: Lightweight comparators enable parallel matching across a DRAM-row bucket; restricting them to a subset of DRAM chips per rank (PIM chips) leads to predictable, high-throughput lookups with low overhead.
- **Rank-level PIM pipelining**: The RLU overlaps key traversal with in-rank comparisons, removing CPU participation and reducing cross-chip communication.
- **Algorithm–hardware co-designed data layout**: A PIM-aware simple hash function coupled with duplication-linked lists, data layout, and lightweight encoding, bound bucket growth and align data with DRAM-PIM structure, enabling both data and pipelined parallelism, and mitigating skew-induced slowdown.

Combined, these innovations address memory boundedness, skew, and low parallel efficiency plaguing state-of-the-art designs. Subarray-level search and RLU-driven key traversal shift data-intensive computation and coordination off the CPU and onto DRAM-resident parallel resources. Row-aligned buckets and skew-aware layout enable synchronous fine-grained subarray parallelism with predictable latency, while rank-level pipelining sustains continuous utilization of the search engine and improves parallel efficiency at the DRAM-rank.

We evaluate SPARQ against CPU, GPU, and PIM baselines. The CPU system is a PowerEdge R750 server running DuckDB [82]; the GPU baseline is an NVIDIA A100 using NVIDIA RAPIDS [22]; and the PIM baselines are PID-Join [53] and SPID-Join [50].

At the operator level, SPARQ delivers order-of-magnitude speedups over the CPU baseline and remains substantially faster than the GPU and prior PIM baselines. On individual SSB SQL queries, running on DuckDB integrated with SPARQ, the join portion average latency decreases by 12× in comparison to CPU DuckDB (Figure 1B), translating to a 5.7× mean end-to-end speedup (1.1–28× per query). Running all SSB queries back-to-back delivers a 4.4× overall query-flight [73] speedup over CPU DuckDB with only 7% data overhead and 2.1% area overhead per PIM chip.

2 Background

2.1 Dynamic Random Access Memory and PIM

Modern main memory systems are dominated by Dynamic Random Access Memory (DRAM), with successive DDR generations (DDR2–DDR5) offering incremental improvements in bandwidth, power efficiency, and capacity [9]. Advanced DRAM modules such as Load-Reduced DIMMs (LRDIMMs) employ on-DIMM buffering to reduce electrical load, enabling higher capacity configurations in server-class systems [34].

DRAM is hierarchically organized into DIMMs, ranks, and banks [58], and further into subarrays and MATs [41]. Each subarray contains a row buffer for fast access to recently activated rows [15, 36, 41, 44, 61]. Within a DIMM, each rank stripes data across multiple DRAM chips to form a 64-bit interface and hide read/write latency with data interleaving and parallel chip activation [58, 62]. DDR4 further controls per-command transfer granularity through the burst length (BL), which specifies how many data transfers occur per command and thus the total data delivered per chip and per rank [57].

PIM aims to reduce CPU-memory data movement by integrating compute near or within memory chips. PIM approaches have explored general-purpose logic inside DRAM [43, 47], including 3D-stacked memories such as HBM and HMC [37, 38], which incur cost, thermal, and scalability challenges [7]. More recent DDR-based PIM systems, such as UPMEM [83], integrate simple RISC cores into each bank for general-purpose applications. When designing a PIM device, strategic placement of processing elements within the memory hierarchy is critical, balancing speed with overhead. From cell array to DIMM level, available internal bandwidth and parallelism decrease, while the ability to increase complexity of computational units increases [56, 61, 85]. Additional challenges include integration complexity and scalability, ensuring compatibility between components, and supporting various problem sizes.

2.2 Relational DBMS and Join Processing

Relational database management systems (DBMSs) store data as relations (tables) containing tuples (rows) and attributes (columns), and execute SQL queries by translating them into physical plans built from operators such as `SELECT/FILTER`, `JOIN`, `GROUP BY`, and `ORDER BY` [66]. Analytical workloads are often organized as star schemas, in which a large fact table captures events and smaller dimension tables provide descriptive context. Queries in this setting typically filter dimension attributes and then join the filtered dimensions with the fact table through key–foreign-key relationships. As a result, JOIN is a critical operator in analytical DBMSs: it enables cross-table correlation and often accounts for a large fraction of end-to-end query latency in decision-support workloads [66], as shown in Figure 1.

Among the different join types (e.g., inner, outer, and cross), hash join is widely used because it is efficient for large, unordered datasets. Hash join has two phases: `BUILD` constructs an in-memory hash table on the build table (usually dimension/smaller table) and organizes entries into hash buckets, while `PROBE` hashes each tuple from the probe table (usually fact/larger table), locates the corresponding bucket, and searches for matching keys inside that bucket. Variants such as Grace hash join, hybrid hash join, and partitioned/parallel hash join adapt this basic algorithm to different memory and parallel-execution settings [66]. When multiple keys map to the same bucket, the implementation resolves collisions through a bucket chain or a probe sequence.

A major challenge in hash joins is data skew, which leads to unbalanced bucket sizes and load imbalance [23]. Uniform hashing is ideal but often infeasible without prior knowledge of the data distribution. To mitigate skew, adaptive strategies such as multi-hashing and dynamic partitioning have been proposed [23].

Another key bottleneck is data movement between CPU and memory [53]. Joins in in-memory databases eliminate disk I/O bottlenecks but remain bottlenecked by memory access: CPUs stall under DRAM bandwidth saturation [42, 55], worsened by increased core counts, Single Instruction Multiple Data (SIMD) widths, and the memory wall. Caches often fail to help due to irregular access patterns, frequent cache misses, and TLB thrashing [55, 76]. NUMA effects can reduce throughput by up to 85% without careful design [3, 4, 42, 48]. Parallelism adds synchronization overhead when accessing shared structures [10, 48], while data skew causes load imbalance [10, 12, 14]. Algorithmic solutions help mitigate these issues [12, 14, 53, 74], but at the cost of overheads such as random memory accesses, additional partitioning phases, and complex skew-handling logic, making performance hardware and dataset dependent [10, 42].

GPUs can outperform CPUs by exploiting massive parallelism and fast memory access in the build and probe phases [78]. However, their effectiveness depends on workload characteristics: join complexity, data locality and transfer patterns, and transfer overhead between host and device. In skewed workloads, GPUs may suffer from severe thread imbalance, sometimes allowing CPUs to match or exceed their performance. Recent work by Chen et al. [14] introduces skew-conscious hash join strategies that dynamically adjust to data distribution, demonstrating that CPUs can outperform GPUs in certain skewed workloads.

PIM can reduce CPU–memory data movement for joins. While prior PIM hash-join designs mainly exploit bank-level parallelism [50, 83], they often require CPU assistance, suffer from capacity limits and internal data movement [53], and provide limited benefit when available parallelism cannot offset PIM's lower operating frequency [39]. As a result, prior join offloading has shown modest gains [39]. However, PIM can deliver large speedups when computation is tightly coupled with memory access; for example, [65] reports **up to 608× speedup** for TPC-H queries using **PIM for non-join** `FILTER` and `AGGREGATE` operations [65].

2.3 Motivation and Characterization

To identify where PIM can most improve analytical-query performance, we first characterize operator time in DuckDB [82] on TPC-H, TPC-DS, and SSB [64]. Across these workloads, **joins account for a large and often increasing fraction of end-to-end query time** as scale grows. In TPC-H, for example, `HASH_JOIN` contributes about 45% of total query time at SF1 and 54% at SF10 on average, reaching 80–98% in several queries. In SSB, joins dominate even more strongly: for join-heavy SF100 Q2.x and Q3.x queries, `JOIN` accounts for roughly 96–99% of execution time. We use SSB for the remainder of our analysis since it represents mainstream behavior of warehouse-scale databases.

The roofline plot in Figure 2 further shows that DuckDB join operations have low arithmetic intensity and performance well below the DRAM-bandwidth roofline, an indicator that join performance is limited primarily by data movement rather than computation.

To understand this bottleneck in more detail, we instrument DuckDB's hash join to isolate performance measurements of joins between the full `lineorder` fact table and each SSB dimension table (`date`, `supplier`, `customer`, and `part`) across multiple scale factors. We collect wall-clock time per build and probe phase, along with more fine-grained probe timing and LLC/dTLB misses normalized by processed rows. We also include two self-joins on the `lineorder` table using the `supplierkey` column: a deduplicated build side table, where each supplier key contributes a single hash-table entry, and an unchanged build side, which captures the impact of build-side duplication. The system setup is in Section 5.

Figure 3(a) shows the fraction of total join time spent in probe and the internal breakdown of probe work. These measurements point to two main lessons. First, **JOIN probe-side execution dominates latency**. For the large SF100 joins, probe accounts for nearly all join time: 99.5% for `part`, 99.7% for `customer`, and 99.3% for the self join. Second, **within probe, hash-table lookup dominates**. Inside the lookup, most of the time is taken by bucket finding and chain walk inside the bucket to find matches.

Figure 3(b) reports LLC and dTLB misses per probe row. The miss breakdown shows that **lookup path is constrained by irregular memory access** rather than useful computation. This rises sharply for larger joins; in particular, dTLB misses dominate for `customer` and `part`, indicating translation-bound lookup.

Figure 3(c) shows probe latency per row increasing from order of 10ns to over 100ns as join size grows, similar to miss intensity.

Figure 3(d) shows effective probe parallelism, defined as thread-sum time divided by wall time. Despite 112 software threads, effective parallelism remains limited (typically below 5×) for most joins, indicating that **memory stalls serialize execution.**

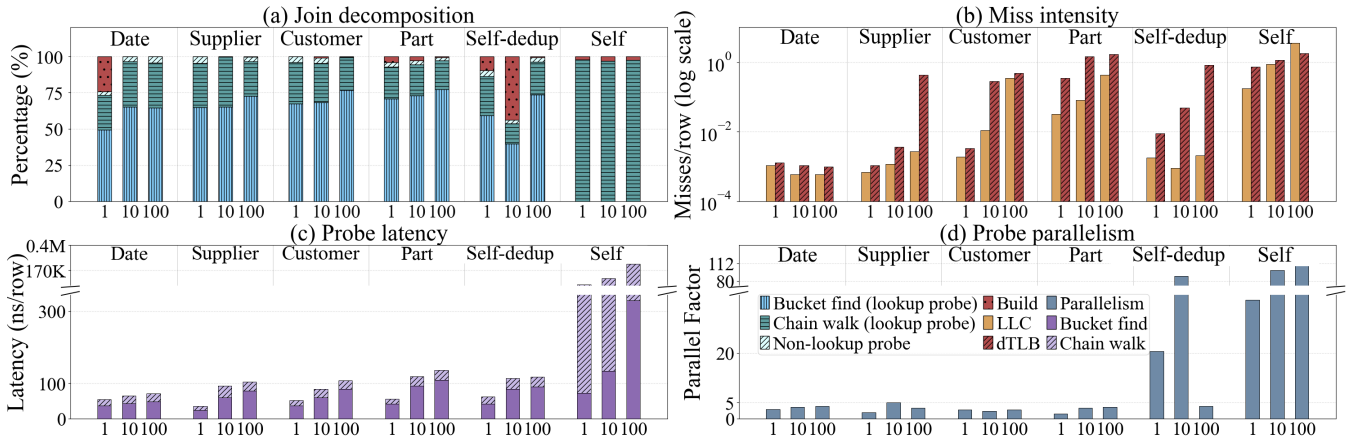


Figure 3: SSB join characterization (X-ax: scale factor). (a) Probe contribution in join and lookup fraction in probe breakdown. (b) LLC/dTLB misses per probe row. (c) Probe cost per row breakdown. (d) Effective probe parallelism (thread-sum / wall time).

In self-join (without deduplication), **heavy build-side duplication causes severe bucket expansion**, shifting the lookup bottleneck from bucket selection to chain traversal. **Chain-walk latency grows** and becomes a major contributor, **increasing the overall probe and join latency by orders of magnitude.** Even with high probe parallelism (36x-109x), **performance remains constrained by irregular memory access**, elevated cache/TLB miss rates, and **longer chains that increase per-thread traversal work.** This effect strengthens with scale: chain-walk latency grows from 40.6K ns/row at SF1 to 100.4K at SF10 and 228.7K at SF100. Over the same range, LLC and dTLB misses increase from 0.1763 to 3.5265 and from 0.7308 to 1.7773 misses/row respectively.

Overall, these results show that CPU hash join is bottlenecked by probe-side random lookup through large pointer-based structures, increasingly dominated by TLB pressure, cache misses, and poor overlap of long-latency memory accesses as hash-table footprint grows. More broadly, the same primitive underlies not only join, but also equality filter, `DISTINCT`, index probe, duplicate detection, and key–value stores lookup, motivating SPARQ's focus on fast, memory-efficient in-memory equality search [29, 40, 66].

3 SPARQ Design

SPARQ accelerates equality-based join and select operators by moving probe into DRAM and supports both conventional memory operation and PIM execution through a mode-switching mechanism controlled by special address commands. At a high level, SPARQ combines three elements: (i) a PIM-enabled DRAM chip that performs subarray-parallel lookup within an activated row, (ii) a rank-level unit (RLU) that orchestrates and pipelines execution across conventional DRAM chips and the PIM chip, and (iii) a duplication-aware layout that keeps the PIM-resident lookup structure compact and predictable. Figure 4 shows the resulting organization, with one PIM-enabled chip per rank. In the PIM setup phase, software constructs a PIM-resident hash structure from the build-side relations, including duplicate handling and dictionary encoding. During join execution, the CPU streams reads of probe keys from the probe table, the RLU pipelines the key fetch with in-memory lookup, and the PIM chip returns matched values. This organization removes CPU-side pointer chasing from the join critical path while preserving a standard DRAM-facing interface.

3.1 Architecture

3.1.1 Subarray-Level Search Engine. SPARQ's core search primitive operates at the subarray level. Given a probe key, the DRAM row containing the target hash bucket in the PIM-enabled chip is activated, the probe key is compared against all stored keys in that row in parallel, and the associated value is returned on a match; if no key matches, a null value is returned.

To support this operation, each PIM-enabled chip adds a set of comparators after the row buffer in each subarray. Each comparator receives one stored key from the activated row buffer and the current probe key. The associated value fields are stored alongside the keys but are not connected to the comparison path. A *Match Select* unit encodes the comparator outputs to identify the matched entry and route its value to the output as shown in Figure 5.

Why subarray-level PIM? A bank-level design reduces the number of comparators, but it also severely limits lookup parallelism. DRAM bank I/O is narrow (e.g., 8 B), whereas a subarray row buffer exposes much larger internal granularity (e.g., 1024 B) [84]. As a result, bank-level search must fetch and scan a bucket incrementally, incurring multiple accesses per lookup. Early lookup termination on the first match makes latency input-dependent and sensitive to skew, while always scanning the full bucket for uniform latency increases unnecessary traffic. Bank-level designs also complicate match selection across multiple accesses and introduce additional serialization. SPARQ compares all candidate entries of full bucket row in parallel, converting slow and variable lookup into a single-row operation with bounded latency.

This choice can incur higher area and layout complexity than bank-level logic because the search logic is distributed across many subarrays rather than shared at the coarser bank level. In addition, subarray-level search operates over a wider row-buffer granularity, requiring more comparison resources per activated row. To bound this overhead, SPARQ enables PIM support in only one chip per

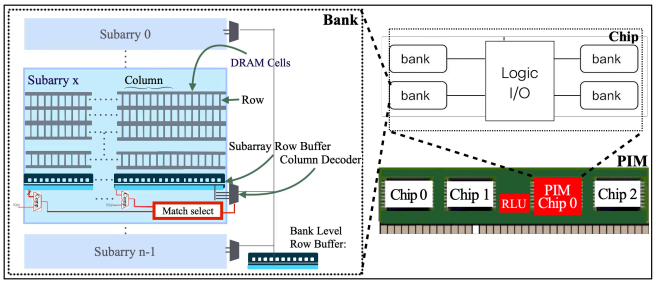


Figure 4: High-level overview of SPARQ architecture.

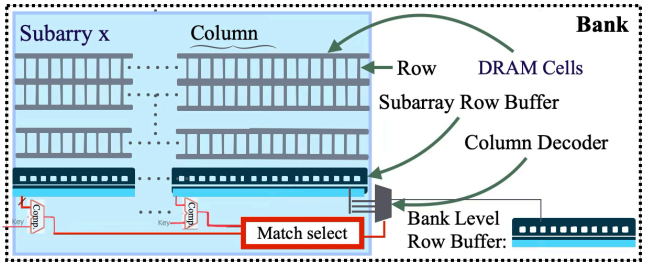


Figure 5: Subarray-level architecture inside a PIM chip.

rank. We assess area overheads in Section 5.3.2. Modern DRAM subarrays are internally partitioned into MATs [31] with shared peripheral structures, so comparator and match-select placement must follow the sensing path rather than an idealized monolithic subarray abstraction as represented in Figure 5.

3.1.2 Rank-Level Unit (RLU). The RLU is the control and pipelining engine of SPARQ. It eliminates repeated CPU-side orchestration by coordinating key fetch from conventional DRAM chips, row activation on the PIM chip, probe-key delivery to the search engine, and result return. The RLU further enables pipelined access to both conventional chips and the PIM-enabled chip within the same rank, allowing SPARQ to overlap probe table key traversal, in-memory lookup, and result return.

SPARQ is invoked from the DBMS through runtime/library calls. When the optimizer detects an equality predicate on a SPARQ-indexed column, the DBMS issues writes to reserved addresses that the RLU interprets as commands, such as `PIM_ON`, `PIM_OFF`, join, select, and update operations. Once enabled, the RLU monitors requests directed to the PIM rank, buffers fetched probe keys, activates the corresponding rows in the PIM-enabled chip, and supplies the probe key to the comparators in the active subarray. On a match, the corresponding value is returned without involving the CPU in per-lookup control.

Chameleon [7] evaluates multiple integration paths for such rank-level logic. In the BCOM-based design (Chameleon-D), the RLU drives addresses through rank-shared RCD control pins [2]; because these lines cannot issue distinct commands to different chips concurrently, accesses to conventional DRAM and the PIM chip are serialized. By contrast, Chameleon-T and Chameleon-S use per-chip DQ paths routed through dedicated LRDIMM data buffers [2, 7], enabling chip-selective command delivery and concurrent access across conventional DRAM and PIM regions. SPARQ follows this latter organization because its performance depends on overlapping probe-key fetch from regular chips with associative lookup on the PIM chip. Moving the RLU's functionality into the memory controller or DMA engine would serialize key traffic over the memory channel and still require explicit CPU coordination, reducing the benefit of in-rank pipelining.

3.1.3 System Interface and Addressing Assumptions. SPARQ reuses the standard DRAM command interface and operates via a kernel driver at the physical-address level, with the OS mapping database pages to a reserved PIM-accessible physical region. SPARQ also assumes deterministic address-to-row mapping in this region with address scrambling disabled like prior PIM designs [47, 51, 75, 80, 85]. Thus, no virtual address translation is required in PIM mode.

3.2 Data Layout

3.2.1 Hash Structure and Duplicate Handling. SPARQ uses a hash join layout tailored to the subarray search engine. Each hash bucket is stored within a single PIM row so that a lookup activates exactly one row and compares all entries in parallel. Unlike CPU joins, which prefer small buckets to remain cache-friendly, SPARQ's large row-sized buckets lookup cost depends on row activation rather than sequential bucket traversal.

To keep bucket size bounded under skew and duplication, SPARQ stores only unique key–value pairs in the PIM-resident hash structure. Repeated occurrences are represented through a separate *duplication-linked list* stored outside the main PIM hash table. Each unique entry in the hash table stores the fixed-size key and an associated value field that either refers to the build-side tuple directly or, in the presence of duplicates, points to the head of a duplication list containing the replicated tuple identifiers. This design decouples duplicate handling from primary lookup, prevents duplicate-heavy buckets from inflating the PIM-resident structure, allows a bounded bucket size, and keeps in-memory search latency predictable.

Algorithm 1 constructs these build-side structures. For a key that appears once, the hash table stores a direct tuple identifier. When the same key appears again, the original direct entry is converted into a duplication entry: the original tuple identifier and the new duplicate are inserted into a duplication list, and the hash-table value is replaced with a tagged pointer to the head of that list. Subsequent duplicates are appended to the same list.

3.2.2 Encoding and Bucket Placement. SPARQ leverages modified DuckDB's dictionary-encoding of join keys to make entry widths uniform and build the PIM-resident hash table. Dictionary encoding replaces join keys whose native representation exceed the PIM key entry format with compact fixed-size codes during the setup phase. When picking the codes, we ensure that keys are unique and track the bucket size to guarantee that there is no overflow.

To map logical buckets to physical storage, SPARQ divides each encoded key into two parts. The *index bits* select the target PIM row, i.e., the hash bucket, while the remaining bits are stored in the columns of that row and connected to the subarray comparators. Each entry stores its value bits alongside the remaining key bits; the value bits bypass the comparison path and are returned only after a match.

Algorithm 1 Hash table with duplication-list construction

Require: Build-side relation B with join key k and tuple identifier id
Ensure: Hash table H, duplication lists L

```
Initialize H and L to empty
for each tuple e ∈ B do
    key ← e.k
    id ← e.id
    if key ∉ H then
        H[key] ← id                                  // first occurrence
    else if H[key] stores a direct tuple id then
        first_id ← H[key]
        Create a new list L[key]
        Append first_id to L[key]
        H[key] ← head pointer of L[key]
    else
        Append id to L[key]
    end if
end for
```

SPARQ adopts a column-oriented layout for the probe-side keys, since join processing primarily accesses columns rather than full tuples [66]. Figure 6(a) shows the end-to-end data placement and flow within a PIM rank. The probe-key column is striped across conventional DRAM chips, while the build-side hash structure resides in the PIM chip. The RLU buffers probe keys and orchestrates lookup, and the duplication-linked list remains in CPU memory.

3.3 Join Operation

3.3.1 Setup Phase. SPARQ targets equality joins and therefore uses a hash-join execution model. During the setup phase, software constructs the PIM-resident hash structure from the build-side relation. This phase performs dictionary encoding, inserts unique key–value pairs into the PIM-resident hash table, and constructs duplication-linked lists for repeated keys. The resulting structure is persistent across queries and can be reused until invalidated.

3.3.2 Join Execution. To execute a join, the CPU streams reads over the contiguous probe-key column. Each burst fetches multiple keys into the DRAM buffer, from which the RLU fills its internal key buffer. The RLU overlaps four operations: (i) reading probe-key blocks from conventional DRAM chips, (ii) buffering keys locally, (iii) activating the target bucket row on the PIM chip and supplying the remaining key bits, and (iv) returning matched values. Figure 6(b) shows this pipeline.

The CPU controls pipeline depth via a bounded number of in-flight requests, determined by RLU buffer capacity, burst length, key width, and DRAM organization. This avoids buffer overflow while keeping the PIM pipeline occupied. Since probe table keys are distributed across ranks, all ranks process disjoint probe streams in parallel. For contiguous keys, address generation and fetch can be offloaded to DMA or handled via vectorized CPU loads (e.g., AVX [33]) without changing the DRAM interface.

If probe table key blocks are not resident in the PIM rank, the CPU or DMA engine can stream them from conventional memory into the rank's DRAM buffers, after which the RLU consumes them in the same pipeline. The cost of this case is discussed in Sec. 5.2.2.

To further reduce traffic, SPARQ suppresses redundant value returns within a sliding window of in-flight probe keys maintained

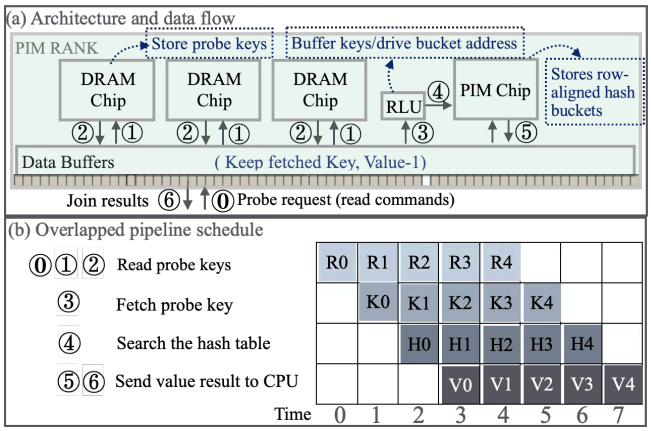


Figure 6: SPARQ end-to-end join execution. (a) Data placement and flow within a PIM rank: probe keys are fetched from conventional DRAM chips, buffered and routed by the RLU, and used to access row-aligned hash buckets in the PIM chip; matching values are returned to the CPU. (b) Overlapped pipeline schedule showing concurrent probe-key fetch, in-memory lookup, and result return.

by the RLU. Repeated keys within this window do not trigger additional value transfers; instead, they reuse the first key's pending or returned result. Keys that recur outside the window are served normally. CPU maintains a matching window of pairs to associate returned matches with the corresponding probe tuples.

3.4 Other Supported Operations

3.4.1 Select Queries. SPARQ also supports equality selects. For a column with an existing SPARQ hash structure, `SELECT distinct` can be served because the PIM-resident structure stores only unique keys. `SELECT where(=)` requires activating a single hash bucket and returning the matched value.

3.4.2 Update Commands. SPARQ supports persistent storage and incremental maintenance of PIM-resident structures. The DBMS can therefore treat SPARQ's datasets similarly to other auxiliary structures such as secondary indexes or materialized views.

Table update. A table update uses burst writes to update a contiguous region in PIM. It takes a base address, a control field, and the updated data payload. Applications may update probe column entries, hash-table entries, or both concurrently. These commands are preferable for initialization and batched modifications. It is mostly used when generating new dataset.

Entry update. An entry update modifies a known PIM-resident location. The DBMS uses it when it already knows the address of the probe-column entry or the hash-table bucket/slot to be changed. Probe-column updates are issued as standard DRAM writes. Hash-table entry updates are sent through reserved PIM command addresses and decoded by the RLU, which writes the payload to the specified bucket/slot without performing a search. This is useful when adding new entries to buckets (a subset of a table update).

Index update. An index update is key-based rather than location-based. The DBMS provides a key and a value, and SPARQ searches

the selected bucket with its comparators to locate the matching entry before updating it. It is useful for incremental updates.

4 Related Work

Recent work shows that PIM can accelerate joins, but early studies caution that simply offloading join operators without PIM-specific co-design may not improve end-to-end performance [39]. Kepe et al. [40] show that PIM benefits depend strongly on operator placement and CPU–PIM scheduling. On UPMEM systems, PIM-Opt [70], PIMDAL [29], and PIMAP [28] expose the communication, orchestration, and scalability limits that arise in data-intensive analytical workloads [13].

Lim et al. [53] evaluate hash join, sort-merge, and nested-loop join on UPMEM and find hash join most PIM-suitable. They introduce PID-Join, a hash join adapted to UPMEM In-DIMM Processors (IDPs). PID-Join [53] maps hash joins onto UPMEM by partitioning input tables, co-locating matching build/probe partitions on the same IDP, and executing each join partition locally. This partitioned execution exploits DPU-level parallelism, but skewed keys create imbalanced partitions and force execution time to be determined by the slowest DPU.

SPID-Join [50] reduces load imbalance across IDPs by replicating join keys across banks and ranks, thereby assigning more compute resources to hot keys. However, this replication increases memory footprint, traffic, and per-IDP work.In both PID-Join and SPID-Join, hash tables are built within UPMEM's WRAM (working RAM or UPMEM's scratch pad memory for its in-DIMM processors). SPID-Join reduces, but does not eliminate, inter-chip communication due to bank-level DRAM capacity limitations and potentially increases UPMEM WRAM overflow.By specializing the PIM datapath and data layout for hash joins, rather than relying on software-managed hash probing on programmable PIM cores, SPARQ overcomes these limitations.

HMC-based PIM studies explore alternative join strategies. Mirzadeh et al. [60] show that sort-merge join can outperform radix hash join on HMC PIM, but do not restructure the join around PIM-specific constraints. Bloom-join designs [25] accelerate conditional joins, yet do not directly address skew or star-schema access patterns common in data warehousing [64, 66]. HMC-based near-memory join designs such as Mondrian [26] also rely on key-based data redistribution before per-partition join execution. This partitioning phase introduces random inter-partition traffic and can become load-imbalanced under skew.

Prior subarray-level techniques expose parallelism via multi-subarray activation or bit-serial/CAM-style engines [15, 44, 51, 77, 80, 85, 86]. In contrast, SPARQ preserves standard DRAM reads, activates only the target subarray, and scans keys in the active row. The RLU orchestrates this primitive into a full join pipeline without memory-controller changes, unlike more hardware-centric near-data designs [71]. Although SPARQ's scan may resemble CAM-style lookup, it differs fundamentally from DRAM-CAM mechanisms [49, 51, 72, 80, 88, 89].SPARQ's comparison logic is placed after the subarray row buffer and operates on a single active row, requiring no changes to the cell array, thereby avoiding the density, energy, and area penalties of CAM cells.

5 Evaluation

5.1 Bottleneck-to-Mechanism Mapping

Using the data in Section 2.3, we map CPU join bottlenecks to SPARQ mechanisms and explain the resulting performance gains.

Probe-side lookup and TLB/LLC pressure. At large scale, join latency is dominated by probe-side lookup. CPU lookup first finds the bucket and then walks chained entries, causing irregular accesses across cache lines and pages. Duplication amplifies this cost: in SF1 self-joins, increasing mean chain length from 4 to 30 raises chain-walk latency from 162.2 to 1058.9 ns/row. SPARQ removes this chain-walk term by mapping each bucket to one DRAM row and searching it with **subarray-parallel comparison**. Its **duplication-aware layout** further bounds lookup by the unique-key structure rather than raw duplicate-chain length.

Parallelism and locality. Figure 3(d) shows that CPU probe parallelism remains below 5× for most joins because dependent, random traversal limits overlap. SPARQ exposes **parallelism across ranks, subarrays, and RLU pipeline** stages. Its **row-aligned layout also improves DRAM locality**: a row miss pays activation, column access, and precharge costs, $t_{\mathrm{RCD}}+t_{\mathrm{CL}}+t_{\mathrm{RP}} \simeq 40ns$ for DDR4, while a row hit avoids activation/precharge and subsequent column commands are limited by $t_{\mathrm{CCD}} \simeq 2.5ns$ [57]. By using row-resident buckets, SPARQ reduces random row activations and shifts lookup toward the lower-latency row-hit path.

Data movement and redundancy. CPU-side probe operations read metadata, follow chains, touch candidate entries, and materialize outputs. SPARQ-side probes operate within a DRAM rank and return only matched values. The **RLU** moves probe orchestration from the CPU into the rank and overlaps key fetch, lookup, and result return. This keeps the in-memory pipeline occupied while reducing CPU involvement and unnecessary movement.

SPARQ also exploits repeated probe keys through CPU–RLU cooperation, suppressing redundant PIM lookups and repeated value returns. At SF100, `lineorder` has about 600M probe rows but only 200K supplier keys, or roughly 3K probes per supplier key. On CPUs, these probes still re-enter the hash-table lookup path and can amplify cache/TLB pressure when the structure is large or chains are long. On UPMEM-style PIM, the same redundancy increases WRAM/MRAM traffic and scratchpad pressure. In SPARQ, repeated keys become an opportunity: nearby duplicates can be coalesced, and repeated accesses to the same row-aligned bucket improve row-buffer locality.

5.2 Experimental Evaluation

5.2.1 Methodology. We evaluate SPARQ using DRAMSim3 [52], modified to model subarray-level activation, row-resident lookup, and trace-driven PIM accesses. In this study, we assume the device is in PIM mode from boot time in order to isolate query-processing latency from software mode-switching overhead. Each RLU includes an 8-entry optimization buffer that coalesces repeated probe keys within a sliding window. The CPU uses the same window to avoid issuing redundant PIM lookups and value returns.

Table 1 summarizes the simulator and host **system configuration**. We use a dual-socket PowerEdge R750. One socket hosts the PIM-enabled DDR4 channels, while the other uses conventional

Table 1: System Configuration Details

Component	Specification
Server Model	PowerEdge R750
CPU	2× Intel Xeon Gold 6330, 56 cores, 2.00 GHz
GPU	NVIDIA A100, 40GB HBM2
Memory	DDR4-3200, 8 channels/socket, 16 DIMMs
Swap Space	200GB Intel Optane NVMe
Operating System	Linux 5.15.0-91-generic
Simulation Tool	DRAMSim3 [52]
Simulated Memory	8 channels of DDR4-3200
Rows and Columns	65,536 rows, 1,024 columns

DDR4 channels. Because each socket accesses its local channels independently, CPU-side memory accesses and PIM operations can proceed concurrently. Under this setup, end-to-end join latency is bounded primarily by the PIM value-return path rather than final result materialization in host memory.

We use SSB **benchmark** [64], which contains the `lineorder` fact table and four dimension tables: `date`, `supplier`, `customer`, and `part`. Datasets are generated with `ssb-dbgen` [24] at SF1–SF100; with the fact table scaling from ≈6M–600M rows. We evaluate individual operators and complete SSB query plans. Complete-query experiments use SF100 and cover four query families: Q1.x filter-heavy queries, Q2.x two-table joins, Q3.x three-table joins with filtering, and Q4.x multi-way joins.

We compare SPARQ against CPU, GPU, and PIM **baselines**. For CPU and GPU runs, all input tables are preloaded in memory before timing to isolate query execution.

5.2.2 Comparison with CPU-Based Systems. The CPU baseline we evaluate against is DuckDB [82], a vectorized analytical DBMS, excelling in in-memory analytics and joins, with optimizations enabled. We also include a single-threaded C++ hash join implementation to isolate the impact of DuckDB's algorithm-level optimizations. Since CPU-based DBMS execution remains dominant, DuckDB is our primary baseline.

Star joins. We first evaluate joins between the `lineorder` fact table and each SSB dimension table. The dimension tables have unique join keys, so the duplication list is empty. Figure 7 compares DuckDB, the single-threaded C++ hash join, and SPARQ. DuckDB improves over the C++ implementation by up to 52×, reflecting the benefit of vectorized execution and CPU-side optimizations. SPARQ improves over DuckDB by more than 400× by moving lookup into DRAM, avoiding CPU-side pointer chasing, reducing cache/TLB misses and random memory accesses, and eliminating unnecessary host-side data movement.

For `customer` and `part`, SPARQ's speedup increases with scale factor as the CPU hash table becomes larger and more memory-bound. The `supplier` join's key distribution creates more repetitions in fact table probes. At SF100, approximately 600M fact rows map to only about 200K supplier keys, with average consecutive repetition lengths of roughly 3K [64]. This repetition reduces PIM row activations and amplifies SPARQ's benefit.

Setup phase. SPARQ materializes its hash structure before query execution and reuses it across joins until invalidated by DBMS. Table 2 reports setup cost for two steps: generating the hash table and duplication-linked list (PIM Generation), and writing the hash and fact-table entries into PIM (PIM Write). We compare these costs with the build phase of a CPU hash join (CPU Build), where

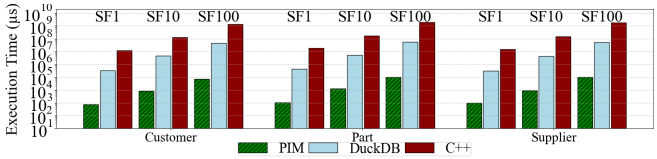


Figure 7: SPARQ vs. CPU-based joins. The duplication list is empty since build table keys are unique. Y-ax is log scale.

Table 2: Join setup latency evaluation (ms).

Table (SF)	CPU Build	PIM Generation	PIM Write
Customer (SF1)	992.75	5.2	0.1545
Part (SF1)	1151	35.6	0.1545
Supplier (SF1)	1007.8	0.3	0.1545
Customer (SF10)	10208	56.4	1.6
Part (SF10)	10791.5	152.0	1.6
Supplier (SF10)	10100.75	3.0	1.6
Customer (SF100)	768192.5	1220.0	11.1
Part (SF100)	773777	785.0	11.1
Supplier (SF100)	35272.25	67.0	11.1

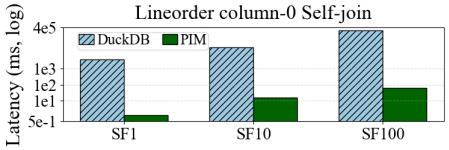


Figure 8: SPARQ vs DuckDB `lineorder` self-join on `col0`.

we partition the tables for multi-threading and build the relevant hash data set. For fair comparison, all PIM/CPU setup/build phases are implemented in optimized single-threaded C++ using memory-mapped I/O. PIM writes are simulated with DRAMSim3 traces that sequentially populate PIM ranks [52].

PIM write latency scales mainly with fact-table size and remains small relative to Build. Hash-table generation is more sensitive to dimension-table size and memory behavior: `part` is expensive because it has the largest dimension table, while `customer` shows a sharp increase at SF100 due to poorer cache locality and collision handling. SPARQ setup is substantially lower than Build and is amortized across repeated queries on the same indexed column.

Self joins and duplication. We evaluate `lineorder` self-joins to stress duplicate-heavy build inputs. In a self-join, repeated keys create quadratic output growth. This makes several columns infeasible to run directly. We focus on `col0`, the only column for which DuckDB completes SF1–SF100 self-joins within our benchmark budget. We use `COUNT(*)` for DuckDB tests to keep the experiment feasible. Thus, DuckDB still builds the hash table and traverses matching chains, but it does not materialize the full join output, instead accumulates the match count during traversal. Figure 8 shows the results. DuckDB latency grows by two orders of magnitude from SF1 to SF100, while SPARQ remains in the millisecond range, yielding hundreds of times speedup.

The benefit comes from SPARQ's duplication-aware layout. DuckDB builds and probes a CPU-resident chained hash table with repeated keys; even with `COUNT(*)`, it must traverse matching chains to count results. SPARQ stores only unique key–value pairs in the PIM-resident hash table and keeps duplicate occurrences in the duplication-linked list. Thus, SPARQ bounds lookup cost by the unique-key structure and returns matched identifiers to the CPU, making self-join execution much less sensitive to duplication.

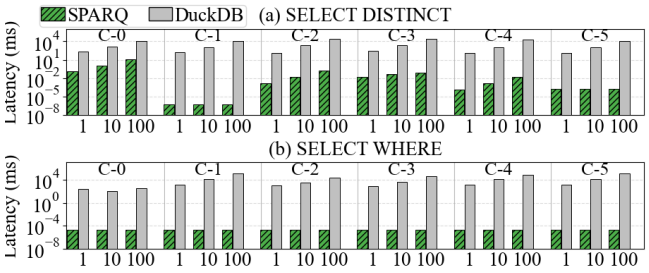


Figure 9: SPARQ vs. DuckDB `SELECT DISTINCT`/ `SELECT WHERE` (=) queries over `lineorder` columns. Y-axis is log scale.

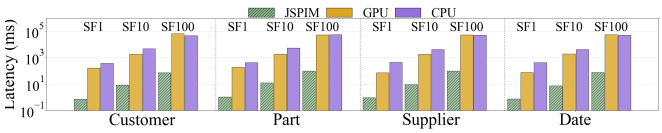


Figure 10: SPARQ vs. GPU/CPU join latency without duplication lists. GPU keeps results on device; SPARQ streams matched results to the CPU. Y-axis is log scale.

Select queries. Figure 9 reports `SELECT DISTINCT` and equality `SELECT WHERE` on `lineorder`. For `SELECT DISTINCT`, DuckDB scans the input column and builds a CPU-side aggregation structure, so latency grows with scale and distinct-cardinality. SPARQ reuses the PIM-resident hash dataset, where unique keys are already materialized, and streams only distinct entries back to CPU.

For equality `SELECT WHERE`, DuckDB still scans the reference column, and latency depends on both input size and output cardinality. At SF100, predicates returning many rows, such as `linenumber=3`, take 137 s, while highly selective predicates such as `orderkey` take only 290 ms. SPARQ performs a single DRAM row-resident bucket lookup, so its latency is nearly constant across scale factors and independent of column size. Overall, SPARQ remains in the ns/ms range, yielding orders-of-magnitude speedups for select operators.

5.2.3 Comparison with GPU-Based Systems. The GPU baseline uses NVIDIA RAPIDS [69] with CUDA [63] and Dask [1, 5] to pipeline/partition large input transfers with task scheduling when spilling is required.

Figure 10 compares SPARQ with the GPU/CPU baselines for fact–dimension joins without duplication lists. GPU measurements exclude query-optimization setup (~0.03s) and CPU–GPU transfer time to isolate record-matching execution. SPARQ outperforms both baselines. GPU is competitive on smaller inputs but loses efficiency as inputs grow and spilling to main memory becomes necessary, while SPARQ continues to operate in place and streams only matched results to the CPU.

5.2.4 Comparison with PIM Baselines. For PIM baselines, we compare SPARQ against SPID-Join [50] and PID-Join [53], two UPMEM-based join accelerators evaluated on a system with 4 memory channels and 16 PIM ranks [50, 53]. We report SPARQ speedups over the results in [50], considering only in-memory join execution and result-transfer-to-host latencies. Since simulators such as DRAMSim3/Ramulator [54] model DRAM timing, banks, command queues, and memory-controller behavior, but not UPMEM's programmable DPU cores, ISA execution, tasklet scheduling, WRAM/MRAM hierarchy, runtime calls, or host–DPU synchronization and UPMEM's general-purpose DPUs (e.g., bank-level parallelism), simulating UPMEM workloads is infeasible [32]. Instead, we reconfigure SPARQ to match the memory channels, ranks, and DRAM timing parameters of SPID/PID, and evaluate it using their workloads. The dataset [50] consists of 32-bit key–value pairs over tables R and S (query $R \bowtie S$), with $|R| \in 0.5\text{M}, 8\text{M}, 32\text{M}$ and $|S|$ initialized to $|R|$ and doubled across four steps per $|R|$. Each configuration sweeps Zipf factors (0, 0.5, 1.5, 2) to model skew.

Table 3: Comparison of PID-Join, SPID-Join, and SPARQ.

Parameter	PID-Join [53]	SPID-Join [50]	SPARQ
System Settings (CMP = Comparator)			
DRAM Type	DDR4 UPMEM	DDR4 UPMEM	DDR4 LRDIMM
Area (per rank)	64×(88KB RAM+PU)	64×(88KB RAM+PU)	1 RLU + 105K CMP
Level	Bank	Bank	Subarray + Rank
Latency Analysis			
Data Sensitivity	High (skew)	Low (distribution+copy)	Low
Data Movement	WRAM transfers	CPU-mediated inter-rank	In-place lookup
Layout Tuning	No	Required	No
Latency Speedup (Zipf 0–2, $R \bowtie S$, $\lvert S\rvert \in [\lvert R\rvert, 8\lvert R\rvert]$)			
$\lvert R\rvert$ = 32M	Baseline	1–3× PID	20–300× SPID
$\lvert R\rvert$ = 8M	Baseline	1–6× PID	15–100× SPID
$\lvert R\rvert$ = 0.5M	Baseline	1–10× PID	100–200× SPID

Table 3 compares SPARQ with PID-Join and SPID-Join. Both baselines target UPMEM, where joins execute as software kernels on simple DPU cores. This improves parallelism over CPUs but still leaves several serialized costs: data must be staged between MRAM and the small WRAM scratchpad, hash probing is performed in software, inter-DPU/rank movement is host-mediated, and skew increases load imbalance and memory pressure. These costs explain why PID-Join is skew-sensitive and why both baselines encounter capacity limits: PID-Join fails at 8M tuples when Zipf ≥ 1.5, and SPID-Join fails at $|R| = 32\text{M}$, $|S| = 64\text{M}$ with Zipf $= 2$.

SPARQ removes these costs: (1) row-resident lookup replaces DPU software traversal with one DRAM-row activation; this directly removes the dominant per-probe software lookup path and provides the largest single gain. (2) SPARQ avoids WRAM staging and CPU-mediated inter-rank redistribution, which removes another multiplicative communication cost when the relation is distributed across PIM ranks. Third, the CPU–RLU collaborative elimination suppresses repeated probe keys, while row-aligned buckets improve row-buffer locality for repeated accesses. Thus, redundancy helps SPARQ by reducing lookup costs and value returns, whereas it hurts baselines by increasing bucket traversal, random memory traffic, and synchronization.

The speedup range reflects these compounding effects. The low end occurs on small or low-skew inputs, where SPID-Join already exploits bank/rank parallelism. The high end appears on large or skewed inputs, where SPARQ combines bounded in-row lookup, no WRAM capacity bottleneck, no host-mediated redistribution, rank-level pipelining, and duplication-aware lookup.

5.2.5 Complete Query Plan Results. To evaluate system-level impact, we compare full SSB query execution in two modes: unmodified DuckDB and a SPARQ-integrated mode that offloads joins to PIM. We model the integration of SPARQ by identifying the exact

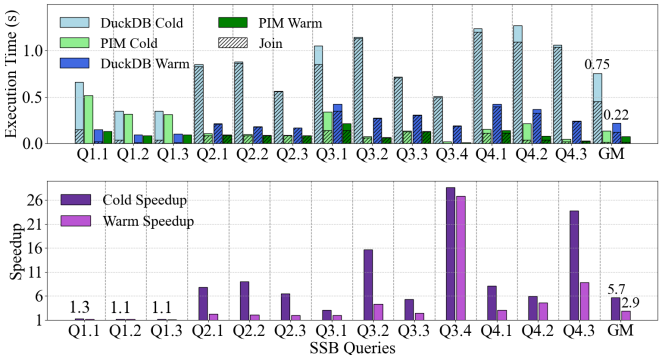


Figure 11: SSB query latency: SPARQ-integrated DuckDB vs. CPU-only DuckDB. Hatched bars indicate join latency.

point in the DuckDB query plan where the join is invoked and replacing that portion with SPARQ's join latency. All data including intermediate result sizes, is taken from DuckDB profiling.

Each query is evaluated under Cold and Warm conditions. Cold runs execute without prior cache warming; Warm runs first execute the same query to expose cache and structure reuse. DuckDB benefits from Warm execution through cached buffers and reused data structures. SPARQ's PIM-side join latency is nearly unchanged between Cold and Warm runs because lookup is performed by PIM rather than CPU caches; therefore, Warm-vs-Cold differences mainly reflect non-join portions of the query.

Figure 11 shows that SPARQ integration improves full-query latency by up to 28×. Q1.x queries see little benefit because they contain little or no join work. Q3.4 achieves the largest speedup because probe-side hash lookups dominate the CPU baseline. In SPARQ, its tiny `date` hash table maps to a compact, row-buffer-friendly PIM layout, replacing irregular CPU probe lookups with low-latency in-DRAM search with row buffer hits and lower PIM–CPU result-transfer cost due to repetitive probe-side keys. Some Q2 and Q4 queries improve less despite having $\geq$ 95% join time in the CPU. Their filtered probe-table cardinality remains high, increasing result-transfer cost. Their larger `part` hash table results in a more scattered PIM layout and higher PIM row buffer misses.

5.3 Overhead and costs

5.3.1 Data Overhead. SPARQ adds the following data structures: a dictionary, copies of encoded fact table key columns, a hash table, and a duplication-linked list. In SPARQ, only the hash table and fact table key column copies are stored in PIM, while the remaining datasets are in CPU memory. When the hash table is small, it can be replicated across DIMMs. For larger datasets, it is distributed across ranks and the fact table is partitioned to align with the relevant hash table segments. In our SSB experiments, with hash table replicated and fact table distributed across ranks, the data overhead is $79.028MB \times SF$, or about 7% of the dataset.

5.3.2 Area Overhead. As a conservative estimate, we assume the RLU's area overhead will be similar to an ARM 28H processor (supporting RLU functionality) with area of $0.020mm^2$ [6], approximately 5.8×10^{-4} of a standard $34mm^2$ DRAM chip [56].

We implemented the subarray-level comparators and match-select units in RTL, then synthesized them using Synopsys Design Compiler in 14*nm* technology [46, 81]. The total area overhead of this design is $726,835.2\mu m^2$ per chip, which is approximately 2.13% of a DRAM chip. While 14*nm* logic transistors are typically denser than DRAM feature sizes (as noted in HiFi-DRAM [56]), practical layout constraints, such as additional wiring, spacing for reliability, and the shared nature of sense amplifiers, may increase the actual overhead by 1.5× to 2× [56]. SPARQ avoids invasive in-DRAM changes by keeping the DRAM cell arrays, bitlines, and external interface intact, adding only lightweight comparators near the row buffer. Unlike prior mechanisms such as DRISA, SALP, LISA, and Sieve [15, 44, 51, 85], SPARQ does not rely on multi-subarray activation or embed general-purpose logic in the subarray periphery, thereby achieving subarray-level parallel search with much lower area and design complexity.

5.3.3 Power Analysis. Adding subarray search logic to SPARQ increases power by up to 0.584μW per active subarray (Synopsys [46]), i.e., $\approx 1.4\times10^{-4}$% of a 408.3mW DRAM-chip baseline [59]. At the rank level (1.63W total [59]), a pessimistically modeled RLU at CPU-like power (6962.34μW from low-power ARM designs [6]) contributes only 0.427%. While the power delivery network must tolerate the added load (similar to Sieve [85]), the compact RLU placed near the LRDIMM buffer reuses existing rails, avoiding new integrity issues, similar to Chameleon [7]. Since SPARQ reduces join latency by up to 100× over CPU baselines, these small power increments translate into substantial net energy savings.

5.3.4 Sensitivity Analysis. As reported in prior work [85] and confirmed by Synopsys, the **subarray-level architecture** adds less than one cycle of latency without affecting standard DRAM timing. However, to assess its impact, we model a fixed per-read delay *t*CMP in DRAMSim3, sweeping it from 0 to 4 cycles while measuring join latency across SSB scale factors. Once the subarray delay exceeds the burst cycle, it becomes a dominant constraint [35] in the controller's scheduling decisions, altering the order of requests and the timing of activate/precharge commands. Increasing *t*CMP from 0 to 1 cycle increases the SSB join operator average latency by 11% (SF1-100). Additional delays beyond this point have diminishing impact, as the pipeline is already stalled, with an average join latency increase of 32% at $t\text{CMP} = 4$ and a maximum of 61% for the customer-lineorder join at SF1. This reduces SPARQ's join speedup over DuckDB from 400x–1000x to 324x–916x. When integrated with DuckDB to run full SSB SF100 queries, the speedup range changes from 1.1x–28x to 1.1x–27x.

We analyze **sensitivity to the number of ranks** with 32 DIMMs, comparing one versus two ranks per DIMM. Performance improves with added ranks due to intra-DIMM parallelism, but gains are sublinear as ranks share bandwidth.

6 Conclusion

We presented SPARQ, an LRDIMM PIM accelerator for join and select queries. SPARQ combines subarray-level parallel search, rank-level pipelining, and a duplication-aware data layout to reduce CPU-side lookup stalls, cross-chip communication, and skew sensitivity. Integrated with DuckDB, SPARQ improves end-to-end SSB query performance by 1.1–28× over CPU-only DuckDB, with 7% data overhead and 2.1% area overhead per PIM-enabled chip.

References

[1] 2020. Dask: Python Parallel Library. https://docs.dask.org/en/stable/. Accessed: 2025-02-05.

[2] 2023. DDR4 Registering Clock Driver Definition (DDR4RCD02). https://www.jedec.org Revision of JESD82-31A, August 2019. Downloaded by Sabiha Tajdari on Dec 12, 2024..

[3] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively parallel sort-merge joins in main memory multi-core database systems. *arXiv preprint arXiv:1207.0145* (2012).

[4] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively parallel sort-merge joins in main memory multi-core database systems. *arXiv preprint arXiv:1207.0145* (2012).

[5] Anton Malakhov. 2016. Composable Multi-Threading for Python Libraries. In *Proceedings of the 15th Python in Science Conference*, Sebastian Benthall and Scott Rostrup (Eds.). 15 – 19. doi:10.25080/Majora-629e541a-002

[6] Arm Limited. 2020. *Arm Cortex-M4 Processor Datasheet.* Datasheet. Arm Ltd. https://developer.arm.com/documentation/102832/latest/

[7] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. 2016. Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems. In *2016 49th annual IEEE/ACM international symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[8] Kazi Asifuzzaman, Narasinga Rao Miniskar, Aaron R. Young, Frank Liu, and Jeffrey S. Vetter. 2023. A survey on processing-in-memory techniques: Advances and challenges. *Memories - Materials, Devices, Circuits and Systems* 4 (2023), 100022. doi:10.1016/j.memori.2022.100022

[9] R. Balasubramonian. 2022. *Innovations in the Memory System.* Springer International Publishing. https://books.google.com/books?id=i4ZyEAAAQBAJ

[10] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment* 7, 1 (2013), 85–96.

[11] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Ozsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 362–373. doi:10.1109/ICDE.2013.6544839

[12] Çağrı Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. 2014. Main-memory hash joins on modern processor architectures. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2014), 1754–1766.

[13] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2023. Adaptive Query Compilation with Processing-in-Memory. In *HardBD/Active 2023*.

[14] Yuzhou Cai and Shimin Chen. 2024. CPU and GPU Hash Joins on Skewed Data. In *2024 IEEE 40th International Conference on Data Engineering Workshops (ICDEW)*. 402–408. doi:10.1109/ICDEW61823.2024.00064

[15] Kevin K Chang, Prashant J Nair, Donghyuk Lee, Saugata Ghose, Moinuddin K Qureshi, and Onur Mutlu. 2016. Low-cost inter-linked subarrays (LISA): Enabling fast inter-subarray data movement in DRAM. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 568–580.

[16] Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. 2007. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)* 32, 3 (2007), 17–es.

[17] Xinyu Chen, Yao Chen, Ronak Bajaj, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2020. Is fpga useful for hash joins?. In *Conference on Innovative Data Systems Research (CIDR)*.

[18] Xuntao Cheng, Bingsheng He, Xiaoli Du, and Chiew Tong Lau. 2017. A study of main-memory hash joins on many-core processor: A case with intel knights landing architecture. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 657–666.

[19] Yannis Chronis, Anastasia Ailamaki, Lawrence Benson, Helena Caminal, Jana Gičeva, Dave Patterson, Eric Sedlar, and Lisa Wu Willis. 2025. Databases in the Era of Memory-Centric Computing. In *Conference on Innovative Data Systems Research (CIDR)*.

[20] Costel Gabriel Corlățan, Marius Mihai Lazăr, Valentina Luca, and Octavian Teodor Petricică. 2014. Query Optimization Techniques in Microsoft SQL Server. *Database Systems Journal* 5, 2 (2014).

[21] Intel Corporation. 2025. Intel® Advisor – Roofline Analysis. https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html. Accessed: 2025-08-17.

[22] NVIDIA Corporation. 2024. RAPIDS: GPU-Accelerated Data Science and Machine Learning. https://rapids.ai/ Accessed: 2024-11-22.

[23] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. 1997. Perfect hashing. *Theoretical Computer Science* 182, 1 (1997), 1–143. doi:10.1016/S0304-3975(96)00146-6

[24] SSB DBGEN. 2024. SSB DBGEN. https://github.com/electrum/ssb-dbgen/blob/master/TPCHREADME Accessed: 2024-11-19.

[25] Sairo R. dos Santos, Francis B. Moreira, Tiago R. Kepe, and Marco A. Z. Alves. 2022. Advancing Database System Operators with Near-Data Processing. In *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 127–134. doi:10.1109/PDP55904.2022.00028

[26] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. 2017. The Mondrian Data Engine. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 639–651. doi:10.1145/3079856.3080233

[27] Jennie Duggan, Olga Papaemmanouil, Leilani Battle, and Michael Stonebraker. 2015. Skew-aware join optimization for array databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 123–135.

[28] M. Frouzakis et al. 2025. PIMAP: Characterizing a Real PIM System for Analytical Processing. In *PACT Poster*.

[29] Manos Frouzakis, Juan Gómez-Luna, Geraldo F. Oliveira, Mohammad Sadrosadati, and Onur Mutlu. 2025. PIMDAL: Mitigating the Memory Bottleneck in Data Analytics using a Real Processing-in-Memory System. *arXiv preprint arXiv:2504.01948* (2025). arXiv:2504.01948 [cs.AR]

[30] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2009. *Database Systems: The Complete Book* (2nd ed.). Pearson.

[31] Heonjae Ha. 2018. *Understanding and Improving the Energy Efficiency of DRAM.* Ph.D. dissertation. Stanford University. https://purl.stanford.edu/yp843xn4828

[32] Bongjoon Hyun, Taehun Kim, Dongjae Lee, and Minsoo Rhu. 2024. Pathfinding Future PIM Architectures by Demystifying a Commercial PIM Technology. arXiv:2308.00846 [cs.AR] https://arxiv.org/abs/2308.00846

[33] Intel Corporation. 2022. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html Volume 2A: Instruction Set Reference, A-L. Order Number: 325383-078US.

[34] Md. Ashraful Islam, Md. Yeasin Arafath, and Md. Jahid Hasan. 2014. Design of DDR4 SDRAM controller. In *8th International Conference on Electrical and Computer Engineering*. 148–151. doi:10.1109/ICECE.2014.7026950

[35] Bruce Jacob, Spencer Ng, and David Wang. 2009. DRAM Memory Access Protocols. https://my.eng.utah.edu/~cs7810/pres/dram-cs7810-protocolx2.pdf Lecture slides.

[36] Bruce Jacob, David Wang, and Spencer Ng. 2010. *Memory systems: cache, DRAM, disk.* Morgan Kaufmann.

[37] Joe Jeddeloh and Brent Keeth. 2012. Hybrid memory cube new DRAM architecture increases density and performance. In *2012 symposium on VLSI technology (VLSIT)*. IEEE, 87–88.

[38] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. 2017. HBM (High Bandwidth Memory) DRAM Technology and Architecture. In *2017 IEEE International Memory Workshop (IMW)*. 1–4. doi:10.1109/IMW.2017.7939084

[39] Tiago R. Kepe, Eduardo C. de Almeida, and Marco A. Z. Alves. 2019. Database processing-in-memory: an experimental study. *Proc. VLDB Endow.* 13, 3 (nov 2019), 334–347. doi:10.14778/3368289.3368298

[40] Tiago R. Kepe, Eduardo C. de Almeida, and Marco A. Z. Alves. 2019. Database Processing-in-Memory: An Experimental Study. *Proceedings of the VLDB Endowment* 13, 3 (2019), 334–347. doi:10.14778/3368289.3368298

[41] Bahareh Khabbazan, Marc Riera, and Antonio González. 2025. Towards Efficient LUT-based PIM: A Scalable and Low-Power Approach for Modern Workloads. arXiv:2502.02142 [cs.AR] https://arxiv.org/abs/2502.02142

[42] 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. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1378–1389.

[43] Jin Hyun Kim, Shin-haeng Kang, Sukhan Lee, Hyeonsu Kim, Woongjae Song, Yuhwan Ro, Seungwon Lee, David Wang, Hyunsung Shin, Bengseng Phuah, Jihyun Choi, Jinin So, YeonGon Cho, JoonHo Song, Jangseok Choi, Jeonghyeon Cho, Kyomin Sohn, Youngsoo Sohn, Kwangil Park, and Nam Sung Kim. 2021. Aquabolt-XL: Samsung HBM2-PIM with in-memory processing for ML accelerators and beyond. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–26. doi:10.1109/HCS52781.2021.9567191

[44] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. 2012. A case for exploiting subarray-level parallelism (SALP) in DRAM. *ACM SIGARCH Computer Architecture News* 40, 3 (2012), 368–379.

[45] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous memory access chaining. (2015).

[46] P. Kurup and T. Abbasi. 1997. *Logic Synthesis Using Synopsys®.* Springer US. https://books.google.com/books?id=dwd5KSVfUVEC

[47] Yongkee Kwon, Kornijcuk Vladimir, Nahsung Kim, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Guhyun Kim, Byeongju An, Jeongbin Kim, Jaewook Lee, Ilkon Kim, Jaehan Park, Chanwook Park, Yosub Song, Byeongsu Yang, Hyungdeok Lee, Seho Kim, Daehan Kwon, Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, Jungyeon Kim, Junyeol Jeon, Myeongjun Lee, Minyoung Shin, Minhwan Shin, Jaekyung Cha, Changson Jung, Kijoon Chang, Chunseok Jeong, Euicheol Lim, Il Park, Junhyun Chun, and Sk

Hynix. 2022. System Architecture and Software Stack for GDDR6-AiM. In *2022 IEEE Hot Chips 34 Symposium (HCS)*. 1–25. doi:10.1109/HCS55958.2022.9895629

[48] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. 2013. Massively parallel NUMA-aware hash joins. In *International Workshop on In-Memory Data Management and Analytics*. Springer, 3–14.

[49] Jinho Lee, Jongwook Chung, Jung Ho Ahn, and Kiyoung Choi. 2017. Excavating the Hidden Parallelism Inside DRAM Architectures With Buffered Compares. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 6 (2017), 1793–1806. doi:10.1109/TVLSI.2017.2655722

[50] 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, Article 251 (Dec. 2024), 27 pages. doi:10.1145/3698827

[51] Shuangchen Li, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. 2017. DRISA: A DRAM-based Reconfigurable In-Situ Accelerator. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 288–301.

[52] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. 2020. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. *IEEE Computer Architecture Letters* 19, 2 (2020), 106–109. doi:10.1109/LCA.2020.2973991

[53] 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 Manag. Data* 1, 2, Article 113 (jun 2023), 27 pages. doi:10.1145/3589258

[54] Haocong Luo, Yahya Can Tuğrul, F. Nisa Bostancı, Ataberk Olgun, A. Giray Yağlıkçı, and Onur Mutlu. 2023. Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator. arXiv:2308.11030 [cs.AR] https://arxiv.org/abs/2308.11030

[55] Stefan Manegold, Peter Boncz, and Martin Kersten. 2002. Optimizing main-memory join on modern hardware. *IEEE transactions on knowledge and data engineering* 14, 4 (2002), 709–730.

[56] Michele Marazzi, Tristan Sachsenweger, Flavien Solt, Peng Zeng, Kubo Takashi, Maksym Yarema, and Kaveh Razavi. 2024. HiFi-DRAM: Enabling High-fidelity DRAM Research by Uncovering Sense Amplifiers with IC Imaging. In *ISCA*.

[57] Inc. Micron Technology. 2023. 8Gb: x4, x8, x16 DDR4 SDRAM. https://www.mouser.com/datasheet/2/671/Micron050920238gbddr4sdram-3175546.pdf Accessed: 2024-11-19.

[58] Micron Technology, Inc. 2000. 512Mb: x4, x8, x16 DDR SDRAM Datasheet. https://www.micron.com Rev. Q; Core DDR Rev. E.

[59] Micron Technology, Inc. 2017. *Calculating Memory Power for DDR4 SDRAM*. Technical Note. Micron Technology. https://www.mouser.com/pdfDocs/tn4007ddr4powercalculation.pdf

[60] Nooshin S Mirzadeh, Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Sort vs. hash join revisited for near-memory execution. In *Fifth Workshop on Architectures and Systems for Big Data (ASBD 2015)*.

[61] Hwayong Nam, Seungmin Baek, Minbok Wi, Michael Jaemin Kim, Jaehyun Park, Chihun Song, Nam Sung Kim, and Jung Ho Ahn. 2024. DRAMScope: Uncovering DRAM Microarchitecture and Characteristics by Issuing Memory Commands. arXiv:2405.02499 [cs.CR] https://arxiv.org/abs/2405.02499

[62] 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 *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 117–130. https://www.usenix.org/conference/atc21/presentation/nider

[63] NVIDIA Corporation. 2023. *CUDA Toolkit Documentation*. https://developer.nvidia.com/cuda-toolkit Version 12.2.

[64] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. 2007. The star schema benchmark (SSB). *Pat* 200, 0 (2007), 50.

[65] Ben Perach, Ronny Ronen, Benny Kimelfeld, and Shahar Kvatinsky. 2024. Understanding Bulk-Bitwise Processing In-Memory Through Database Analytics. *IEEE Transactions on Emerging Topics in Computing* 12, 1 (2024), 7–22. doi:10.1109/TETC.2023.3315189

[66] Hasso Plattner et al. 2013. *A course in in-memory data management*. Springer.

[67] H. Plattner and A. Zeier. 2012. *In-Memory Data Management: Technology and Applications*. Springer Berlin Heidelberg. https://books.google.com/books?id=HySCgzCApsEC

[68] SSB QUERIES. 2024. SSB QUERIES. https://github.com/implydata/benchmark-tools/blob/master/StarSchemaBenchmark/AllSSBQueries.txt Accessed: 2024-11-19.

[69] RAPIDS Development Team. 2020. RAPIDS: Open GPU Data Science. https://rapids.ai/. Accessed: 2025-02-05.

[70] T. Rhyner et al. 2024. PIM-Opt: Understanding Distributed Optimization on PIM. In *PACT*.

[71] Yesin Ryu, Yoojin Kim, Giyong Jung, Jung Ho Ahn, and Jungrae Kim. 2024. Native DRAM Cache: Re-architecting DRAM as a Large-Scale Cache for Data Centers. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1144–1156. doi:10.1109/ISCA59077.2024.00086

[72] Yesin Ryu, Yoojin Kim, Giyong Jung, Jung Ho Ahn, and Jungrae Kim. 2024. Native DRAM Cache: Re-architecting DRAM as a Large-Scale Cache for Data Centers. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1144–1156. doi:10.1109/ISCA59077.2024.00086

[73] Jimi Sanchez. 2016. A Review of Star Schema Benchmark. arXiv:1606.00295 [cs.DB] https://arxiv.org/abs/1606.00295

[74] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1961–1976. doi:10.1145/2882903.2882917

[75] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2017. Ambit: in-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 273–287. doi:10.1145/3123939.3124544

[76] Ambuj Shatdal, Chander Kant, and Jeffrey F Naughton. 1994. *Cache conscious algorithms for relational query processing*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[77] Akhil Shekar, Morteza Baradaran, Sabiha Tajdari, and Kevin Skadron. 2023. HashMem: PIM-based Hashmap Accelerator. arXiv:2306.17721 [cs.AR] https://arxiv.org/abs/2306.17721

[78] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 698–709. doi:10.1109/ICDE.2019.00068

[79] Utku Sirin and Anastasia Ailamaki. 2020. Micro-architectural analysis of OLAP: limitations and opportunities. *Proc. VLDB Endow.* 13, 6 (Feb. 2020), 840–853. doi:10.14778/3380750.3380755

[80] Melina Soysal, Konstantina Koliogeorgi, Can Firtina, Nika Mansouri Ghiasi, Rakesh Nadig, Haiyu Mao, Geraldo Francisco de Oliveira Junior, Yu Liang, Klea Zambaku, Mohammad Sadrosadati, and Onur Mutlu. 2025. MARS: Processing-In-Memory Acceleration of Raw Signal Genome Analysis Inside the Storage Subsystem. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS '25)*. Association for Computing Machinery, New York, NY, USA, 513–534. doi:10.1145/3721145.3730428

[81] Synopsys. 2024. Cloud EDA: Unlimited Access to EDA Software Licenses. https://www.synopsys.com/cloud.html Accessed: 2024-09-19.

[82] DuckDB Development Team. 2024. DuckDB Documentation. https://duckdb.org/docs/index. Accessed on: March 5, 2024.

[83] The Next Platform. 2019. *Accelerating Compute By Cramming It Into DRAM*. https://www.upmem.com/nextplatform-com-2019-10-03-accelerating-compute-by-cramming-it-into-dram/ Accessed: 2024-07-31.

[84] Yaohua Wang, Lois Orosa, Xiangjun Peng, Yang Guo, Saugata Ghose, Minesh Patel, Jeremie S Kim, Juan Gómez Luna, Mohammad Sadrosadati, Nika Mansouri Ghiasi, et al. 2020. Figaro: Improving system performance via fine-grained in-dram data relocation and caching. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 313–328.

[85] Lingxi Wu, Rasool Sharifi, Marzieh Lenjani, Kevin Skadron, and Ashish Venkat. 2021. Sieve: Scalable In-situ DRAM-based Accelerator Designs for Massively Parallel k-mer Matching. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 251–264. doi:10.1109/ISCA52012.2021.00028

[86] Lingxi Wu, Rasool Sharifi, Ashish Venkat, and Kevin Skadron. 2022. DRAM-CAM: General-purpose bit-serial exact pattern matching. *IEEE Computer Architecture Letters* 21, 2 (2022), 89–92.

[87] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.

[88] Keng-Hao Yang, Hsiang-Jen Tsai, Chia-Yin Li, Paul Jendra, Meng-Fan Chang, and Tien-Fu Chen. 2017. eTag: Tag-Comparison in Memory to Achieve Direct Data Access based on eDRAM to Improve Energy Efficiency of DRAM Cache. *IEEE Transactions on Circuits and Systems I: Regular Papers* 64 (2017), 858–868. https://api.semanticscholar.org/CorpusID:2284234

[89] Leonid Yavits. 2023. DRAMA: Commodity DRAM based Content Addressable Memory. arXiv:2312.15527 [cs.AR] https://arxiv.org/abs/2312.15527