Accepted: 07 December 2025
Revised: 24 November 2025
Received: 14 September 2025

RESEARCH-ARTICLE

Membrane: Accelerating Database Analytics with DRAM-Based PIM Filtering and Schema Denormalization

AKHIL SHEKAR, University of Virginia, Charlottesville, VA, United States

KEVIN P GAFFNEY, University of Wisconsin-Madison, Madison, WI, United States

MARTIN PRAMMER, Carnegie Mellon University, Pittsburgh, PA, United States

KHYATI KIYAWAT, University of Virginia, Charlottesville, VA, United States

LINGXI WU, University of Virginia, Charlottesville, VA, United States

HELENA CAMINAL, Cornell University, Ithaca, NY, United States

View all

Membrane: Accelerating Database Analytics with DRAM-Based PIM Filtering and Schema Denormalization

AKHIL SHEKAR, Dept of Computer Science, University of Virginia, Charlottesville, United States
KEVIN GAFFNEY, Dept of Computer Science, University of Wisconsin-Madison, Madison, United States
MARTIN PRAMMER, Dept of Computer Science, Carnegie Mellon University, Pittsburgh, United States
KHYATI KIYAWAT, Dept of Computer Science, University of Virginia, Charlottesville, United States
LINGXI WU, Dept of Computer Science, University of Virginia, Charlottesville, United States
HELENA CAMINAL, Cornell University, Ithaca, United States
ZHENXING FAN, University of Virginia, Charlottesville, United States
YIMIN GAO, Dept of Computer Science, University of Virginia, Charlottesville, United States
ASHISH VENKAT, Dept of Computer Science, University of Virginia, Charlottesville, United States
JOSE MARTINEZ, School of Electrical and Computer Engineering, Cornell University, Ithaca, United States
JIGNESH PATEL, Dept of Computer Science, Carnegie Mellon University, Pittsburgh, United States
KEVIN SKADRON, University of Virginia, Charlottesville, United States

In-memory database query processing frequently involves substantial data transfers between the CPU and memory, leading to inefficiencies due to the Von Neumann bottleneck. Processing-in-Memory (PIM) architectures offer a viable solution to alleviate this bottleneck. In our study, we employ a commonly used software approach that streamlines JOIN operations into simpler selection or filtering tasks via pre-join denormalization, thereby making the query processing workload more amenable to PIM acceleration. This research explores the DRAM design landscape to evaluate how effectively these filtering tasks can be executed efficiently across the DRAM hierarchy and their effect on overall application speedup. We also find that operations such as aggregates are better executed on the CPU than on PIM. Thus, we propose a cooperative query processing framework that capitalizes on both CPU and PIM strengths, where (i) the DRAM-based PIM block, with its massive parallelism, supports scan operations while (ii) CPU, with its flexible architecture, supports the rest of the query execution. This allows us to utilize both PIM and CPU where appropriate and prevent dramatic changes to the overall system architecture.

With these minor modifications to the system architecture and a customized version of the DuckDB database to integrate offloaded scan operations into the CPU-side processing, our methodology enables accurate end-to-end performance evaluations using established analytical benchmarks such as TPC-H and the Star Schema Benchmark (SSB). Our findings show that this

Authors' Contact Information: Akhil Shekar, Dept of Computer Science, University of Virginia, Charlottesville, Virginia, United States; e-mail: as8hu@virginia.edu; Kevin Gaffney, Dept of Computer Science, University of Wisconsin-Madison, Madison, Wisconsin, United States; e-mail: kpgaffney@wisc.edu; Martin Prammer, Dept of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, United States; e-mail: mprammer@andrew.cmu.edu; Khyati Kiyawat, Dept of Computer Science, University of Virginia, Charlottesville, Virginia, United States; e-mail: vyn9mp@virginia.edu; Lingxi Wu, Dept of Computer Science, University of Virginia, Charlottesville, Virginia, United States; e-mail: lw2ef@virginia.edu; Helena Caminal, Cornell University, Ithaca, New York, United States; e-mail: hc922@cornell.edu; Zhenxing Fan, University of Virginia, Charlottesville, Virginia, United States; e-mail: fjy3ws@virginia.edu; Yimin Gao, Dept of Computer Science, University of Virginia, Charlottesville, Virginia, United States; e-mail: yg9bq@virginia.edu; Ashish Venkat, Dept of Computer Science, University of Virginia, Charlottesville, Virginia, United States; e-mail: venkat@virginia.edu; Jose Martinez, School of Electrical and Computer Engineering, Cornell University, Ithaca, New York, United States; e-mail: martinez@cornell.edu; Jignesh Patel, Dept of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, United States; e-mail: jignesh@cmu.edu; Kevin Skadron, University of Virginia, Charlottesville, Virginia, United States; e-mail: skadron@virginia.edu.

ACM 1544-3973/2025/12-ART
https://doi.org/10.1145/3786775

novel mapping approach improves performance, delivering a $5.92x/6.5x$ speedup (TPCH/SSB) compared to a traditional schema and $3.03 - 4.05x$ speedup compared to a denormalized schema with $9 - 17\%$ memory overhead, depending on the degree of partial denormalization. Further, we provide insights into query selectivity, memory overheads, and software optimizations in the context of PIM-based filtering, which better explain the behavior and performance of these systems across the benchmarks.

CCS Concepts: • **Computer systems organization** → *Other architectures.*

Additional Key Words and Phrases: Processing-in-Memory, DRAM, Databases

1 Introduction

Online Analytic Processing (OLAP) systems are critical technologies used to unlock the potential of vast enterprise databases. These systems employ analytic SQL queries to transform database contents into graphs on live dashboards, generate summary reports for key performance indicators (KPIs), and trigger alerts when KPIs deviate from the norm. In modern enterprise settings, such analytic SQL queries are also used to prepare enterprise databases for downstream machine learning (ML) pipelines (i.e., the data-heavy portions of an ML pipeline related to data cleaning and feature engineering are often done in SQL).

Enterprise databases have consistently grown in size over the past five decades. Historically, Moore's Law allowed hardware performance to keep up, while maintaining a near-constant cost from one hardware generation to the next. However, it is now evident that this trajectory is no longer sustainable. Indeed, Google recently showed results from profiling its fleet and found that BigQuery, an analytics platform, consumed about 10% of total cycles within the fleet, and proposed analytics as a candidate for acceleration. [24]

Furthermore, the importance of *in-memory* database organizations is growing rapidly for OLAP systems, including in data science and business analytics settings where complex analytic queries are often performed with a human-in-the-loop (a key driver behind the rise of DuckDB) [4], requiring high performance on individual queries, in addition to high overall throughput. Even when the database does not fit in memory, smart methods of caching or staging data from disk are used by the database management system (DBMS) to keep hot data in memory. However, these workloads are often bound by the memory system's performance in conventional von Neumann-style processing systems (which dominates the server landscape on which database systems are deployed) [55]. This memory wall [61] is likely to become worse over time, as memory densities are likely to grow faster than memory bus speeds (both latency and throughput impact OLAP workload performance) [3]. Thus, the memory system is critical for overall query performance [5].

Our paper explores near-data processing and processing-in-memory (PIM) for analytic SQL queries. Notably absent from prior efforts in this domain is an exploration of the different options for placing processing at different locations within the memory architecture, in light of their impact on *end-to-end query execution time*. We explore placing processing elements in the channel interface and the rank, bank, and subarray levels of the memory hierarchy. We find that aggressive PIM architectures are *not* needed, because even modest, bank-level PIM architectures are able to significantly accelerate the PIM-friendly task of filtering the database to find the desired records—enough to make the remaining, less PIM-friendly tasks (fetching the selected records and post-processing, i.e., aggregation, sorting, etc.) the new bottleneck. Further improvements in filtering are bound by Amdahl's Law.

We propose a PIM design that is specialized for data analytics, and filtering in particular, because this represents "low-hanging fruit" for an initial PIM product. Because filtering is so important and primarily involves simple operations on numeric and dictionary-encoded columns, the implementation can achieve high utilization of the new hardware. Our proposed architecture is very lightweight, incurring minimal changes to the DRAM and CPU architecture, and minimal area and power overhead. Indeed, our proposed design adds only an area-optimized comparison unit to each DRAM bank and a small change in how cache line fetches and interleaving interact,

without requiring any other changes to data layouts. It incurs a marginal 0.1% area overhead, avoids the need to restructure data between PIM access and regular memory read/write, and has no impact on conventional read/write performance. This allows the PIM feature to be virtually invisible to applications or application phases that do not use PIM. Our experiments do show that the main drawback of adding PIM is that activating all banks in parallel leads to a 4x increase in DRAM power, requiring improved power delivery and cooling. However, end-to-end energy efficiency improves by 3.4x.

With OLAP, the SQL interface allows the PIM product to be transparent to the users, and the analytics market is large enough that it can likely support and benefit from a specialized PIM product. This avoids the typical chicken-and-egg problem that faces many accelerators, in which there is a lack of applications and programming models to create a ready market. Overall, our goal is a design that can enable low-risk adoption of PIM in commodity DRAM, so that if this design is successful, it can serve as a starting point for more sophisticated and general-purpose PIM architectures.

In this paper, we show that end-to-end query processing does indeed benefit from PIM and present the following contributions:

(1) We concentrate on DRAM-based PIM and explore the hardware design possibilities for moving query processing closer to the data in memory. We argue that the filtering step is both the most important and also the best fit for PIM. The options we consider are rank-level processing (via a small module on the DIMM module's circuit board), two forms of bank-level PIM, and subarray-level PIM, and evaluate their impact on end-to-end performance as well as performance relative to the extra area required to implement them. We show that the bank level provides the best combination of performance and low overhead.

(2) Inspired by a prevalent database technique called WideTable [42], we use denormalization and dictionary encoding to replace joins with filters, improving PIM amenability. Because full denormalization incurs prohibitive space overhead (73% for SSB and exceeding available memory for TPC-H), we propose an approach that uses static analysis of the workload to determine which columns to denormalize. Exploring the tradeoff between space overhead and performance improvement, we find that partial denormalization with PIM filtering enables 5.9x / 6.4x speedup with only 17% / 13% additional space for SSB / TPC-H.

(3) We explore several dimensions of the hardware and software co-design space, and we present a variety of insights on the relationships between hardware parallelism, filter selectivity, database size, software optimization, and performance.

(4) We describe the full end-to-end implementation in DuckDB, a widely-used state-of-the-art OLAP database system [50], including system integration.

2 Background

2.1 OLAP database systems

This paper focuses on accelerating database analytics, in particular online analytical processing (OLAP), a workload category concerned with efficiently extracting insights from large datasets. To facilitate understanding, we provide a brief overview of the aspects of OLAP database systems that are most relevant to our contributions.

2.1.1 Database organization. OLAP databases typically contain a vast amount of historical data that has accumulated over time. This data is typically organized into one or more large, central *fact* tables and several smaller *dimension* tables as shown in Figure 1. Fact tables store the primary entities in the database. For example, an e-commerce company may have an `orders` fact table with one record for each purchased item, including the price, discount, and order date. Dimension tables store additional information about rows in the fact tables. For example, `product` and `customer` dimension tables may contain information about the product that was ordered and the customer that placed the order. OLAP queries typically involve filtering the records of the fact tables and dimension tables, joining the filtered records together, and then grouping, aggregating, and/or sorting the

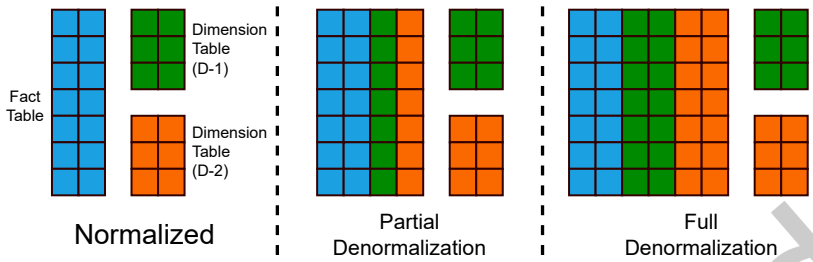


Fig. 1. Normalized vs. Denormalized forms in Database Organization

joined records to produce an informative result. For example, an OLAP query could be used to answer a question of the form, "For each product in category X, what is the maximum discount applied to an order placed by a customer living in city Y?" Tables generally consist of integer, decimal, and string data, which are sometimes combined to form more complex data types. Frequent strings are often dictionary-encoded into integers, and comparisons involving strings use the encoded values when appropriate. Traditional tree-based and hash-based indexes are scarce in OLAP databases due to their space and maintenance overhead, especially with a variety of queries. Instead, lightweight batch-level statistics (*e.g.*, minimum and maximum in DuckDB) are used to improve filter efficiency [1]. Databases targeting analytics are typically laid out in memory in a column-store format, in which a given column (i.e., record field) is laid out consecutively, instead of the traditional row-store, in which all fields of a record are kept together.

2.1.2 Core operators. OLAP database systems employ a small number of logical operators that can be combined to execute complex queries. The input of each operator is one or more logical tables, often referred to as relations in the context of relational algebra. The output is a single logical table. We emphasize the distinction between logical and physical here, noting that the results of each operator are not necessarily materialized and are often streamed to subsequent operators in a pipelined fashion. The *filter* operator (also known as *selection*) returns the subset of its input rows for which some Boolean expression evaluates to true. Filters are often evaluated as part of a table scan, although scans without filters do occur. The *projection* operator computes an expression on its input columns (*e.g.*, multiplying two columns together). The *join* operator finds matching rows between two tables based on some condition. The *aggregation* and *group-by aggregation* operators compute aggregate values (*e.g.*, sum) for one or more groups in the input. Each logical operator has one or more physical implementations that may be specialized for a particular situation. For example, joins with equality conditions are typically executed using hash joins. Operator specialization exposes a natural entry point for integrating PIM filtering into the rest of the database system stack. We propose PIM filtering as a specialization of the filter operator. We provide additional details about system integration later in the paper.

2.1.3 Denormalization. Given the read-mostly and append-only nature of OLAP databases, a common method to speed up query processing is to *denormalize* the database as shown in Figure 1. This technique folds information from the dimension table(s) into the fact table so that a join is no longer needed to evaluate OLAP queries. In research, it has already become a common requirement for software-based OLAP acceleration methods [21, 41, 42, 49, 54]. Denormalization is now prevalent in multiple commercial products as well (*e.g.*, [18, 27, 56]). WideTable [42] is a specific, widely-used style of denormalization. Although denormalization comes at the cost of increasing the database size, dictionary-based encoding can limit this overhead (to 9-17% in our experiments) [15, 22, 34, 42, 51].

In practice, denormalization is applied as part of the database's load-time Extract-Transform-Load (ETL) pipeline. During this process, attributes from the dimension tables are merged into the fact table to create a

wide, join-free layout. This transformation is performed once for each data load or refresh and the resulting denormalized columns are materialized and stored on disk. When the database is subsequently loaded into memory, the system reads this already-denormalized representation directly, incurring no additional per-query overhead. Thus, while denormalization increases the on-disk and in-memory footprint (mitigated by dictionary encoding), it does not introduce query-latency penalties, since all queries operate on the preconstructed wide schema without requiring runtime expansion or join processing. This load-time materialization is standard practice in both research systems and commercial OLAP engines, and is a key reason why denormalization is widely adopted to accelerate analytical workloads.

2.1.4 Memory performance. OLAP queries are data-intensive, involving relatively few processor cycles per byte of input data. For example, when a query asks for all customers in a given zip code, it may scan an entire table while only applying a simple comparison operation on each input record. As CPU speed and memory size have increased faster than both the memory speed and memory bus bandwidth, OLAP query evaluation in main-memory environments (the focus of this paper) is often memory-bound [55].

2.2 DRAM

DRAM exists in multiple configurations, including DDR (typically as DIMMs), and higher-performance variants such as GDDR and HBM. DDR remains the standard for main memory in server systems targeting OLAP workloads. CPUs interface with one or more 64-bit DDR memory channels, each managed by an on-chip memory controller. Channels operate independently, enabling parallel read/write operations. Mainstream CPUs support at least two channels, while server-grade processors may support up to eight. Each channel comprises multiple ranks—groups of DRAM chips operating in parallel. In an x8 configuration, each chip contributes 8 bits toward a 64-bit word, requiring 8 chips per rank. All ranks share the channel's bus, though only one rank can be active at a time. A channel typically supports up to four ranks.

Within each chip, memory is divided into banks, which can be addressed independently but share the internal datapath, limiting simultaneous transfers. Commands to different banks may be pipelined to exploit bank-level parallelism. A logical bank spans all chips in a rank; each chip's portion is a physical bank. Typical DRAM chips contain 8–16 such banks.

Banks are further divided into subarrays, selected via high-order row address bits, with the remaining bits selecting the row within the subarray. Each subarray includes dedicated decoders and a row buffer but shares buses and a global data line (GDL), allowing only one subarray access at a time. Standard access requires precharging the bank, activating the row, decoding the column, and transferring data to the output pins via the GDL. In open-page mode, the row buffer retains data, allowing low-latency successive accesses (t_{CCD}: 4–8 cycles). Accessing a different row requires full activation latency ($t_{RAS} + t_{RCD} + t_{CL}$), typically around 100 cycles.

3 Mapping Data Analytics to PIM

3.1 PIM architecture requirements for data analytics

The landscape of near-data and in-memory processing is extensive, and our principal objective is to optimize specifically for *in-memory* analytics workloads. Given that the data are already in memory, we seek a solution that can operate on the data in place, allowing processing in memory and regular load/store access to the same data, without the need to move data between PIM-friendly and CPU-friendly layouts. We also seek a solution with sufficiently low overhead to justify inclusion within a commodity (albeit premium) DRAM product.

While many PIM designs have been proposed in the past, we look for inspiration to three major commercially-announced PIM systems that respect the constraints mentioned earlier: Samsung's Aquabolt (implemented in HBM) [37], SK hynix's AiM (based on GDDR) [35], and UPMEM (DDR) [25].

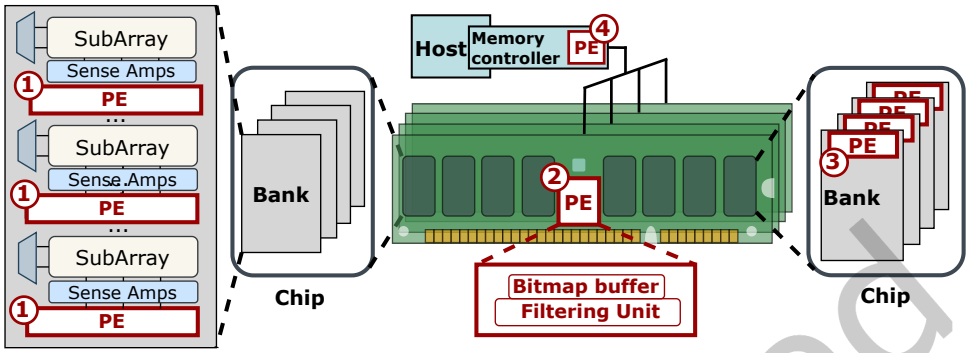


Fig. 2. Membrane Design Space Exploration: ① Processing Element (PE) at the Subarray-level ② PE at the Rank-level ③ PE at the Bank-level ④ PE at the Channel-level

All three extant PIM products place logic at the bank interface, but with significant hardware overhead. Aquabolt and AiM target acceleration of neural network kernels such as GEMV and support SIMD arithmetic units at the bank interface, with significant impact on capacity per unit area, 50% and 20-25% respectively. However, servers designed for in-memory databases seem unlikely to adopt HBM, and currently use traditional SDRAM DIMMs, because this technology is much lower cost and scales much more easily to the sizes needed. UPMEM, on the other hand, implements independent tasklet-based processing at the DDR's bank level, using a 64KB scratchpad per bank and data processing units (DPUs) that can operate independently, in a task-parallel manner. We have not been able to find information about the area overhead of this approach.

We also considered subarray-level PIM (placing units at some or all of the subarrays in each of the banks), rank-level near-data processing (placing units on a DIMM module, not in DRAM chips), and channel-level filtering (placing units at the channel interface just before the cache hierarchy) as shown in Figure 2, and will show that the bank-level approach provides the best balance of performance vs. overhead.

3.2 PIM Amenability Tests

Prior work [7] studies the HBM-PIM [37] style architecture and proposes four PIM-Amenability Tests to assess whether a kernel is suitable for PIM acceleration. The work proposes that a kernel should pass all four tests and not just a subset of them to be well-suited for PIM. Table 1 shows how the three major stages of database analytics, filter, joins, and aggregation, map to these criteria. Only filtering meets all four criteria.
The four major criteria that the test suggests are as follows:

(1) Is the workload memory-bound? Bandwidth inside the DRAM is much higher than the bandwidth of the DRAM interface. If the workload is memory bound and can effectively use this higher internal bandwidth, then it is suitable for PIM acceleration. Otherwise, PIM may save energy but is less likely to boost performance.
(2) Does the workload have low cache reuse? If not, better performance is typically achieved via CPU computation, which operates at a much higher clock speed and benefits from cache reuse.
(3) Are computations localized within a single bank? Transfers between banks or ranks are costly.
(4) Does the workload exhibit memory-aligned data parallelism? PIM architectures that leverage bank and/or subarray-level parallelism compute simultaneously on the same row and column addresses across banks/subarrays. Thus, data must be aligned to be executed in lockstep across multiple banks. Furthermore, this type of data parallelism maximizes row buffer locality.

We would suggest another consideration related to item 4 above: Does the algorithm exhibit sufficient parallelism and operate on large enough data objects to leverage sufficient internal parallelism of the DRAM to show speedup over near-data processing outside the DRAM?

PIM-Amenability Test	Filter	Aggregates	JOINs
Memory-bound?	✔	✔	✔
Low cache-reuse?	✔	✘	✘
Localized operand interaction?	✔	✘	✘
Aligned Data Parallelism?	✔	✘	✘
Run on →	PIM	CPU	CPU

Table 1. Major kernels used in Analytics Database Workloads and their PIM Amenability characteristics.

The filter kernel is memory-bound, because it does not exhibit temporal locality: it streams through the entire table, and elements that do not match the filter predicate are not touched again. Column-oriented schema do exhibit spatial locality, but because the computation density per word is low (just a simple comparison), memory access remains the bottleneck. Typical in-memory databases are many GB in size, and filtering is also embarrassingly parallelizable, allowing full use of the DRAM's internal parallelism, and filtering exhibits aligned data parallelism.

Joins, although memory-bound [17], benefit from having caches while performing hash-join. The join algorithms are tweaked in many instances [9, 11, 53] to make the join algorithm cache-aware, and in many cases, the dimension tables used to create the hash table for the hash join are small enough to fit easily in the CPU cache hierarchy. We also observe that performing a hash join in DRAM would likely require a copy of the hash table in each bank to avoid cross-bank interactions, although this could also be stored in the bank itself, and hashing typically entails random accesses to the hash table, inhibiting aligned data parallelism. Hence, join kernels do not meet 3 out of the 4 PIM-amenability criteria. Furthermore, in comparison to joins, filtering requires only a simple comparison per predicate, instead of hashing plus table lookup, and denormalization is able to convert joins to simple, PIM-friendly, streaming comparison operations, without the complexity of expensive hashing hardware.

Aggregation (grouping, sorting, etc.) is only performed on the selected records. It exhibits greater temporal cache locality and involves more complex computation that would be difficult to localize within a bank and would require more costly processing units in the PIM.

PID-Join [43] explored the idea of using UPMEM [25] bank-level architecture to accelerate the join algorithms. The overall end-to-end query execution was accelerated by only 1.14x despite leveraging the large parallelism and bandwidth increases available at the bank-level. In contrast, our approach emphasizes denormalization as a strategy to reframe the workload itself to be more PIM-compatible. This enables more substantial performance gains while incurring minimal hardware overhead.

Our results, shown in the upcoming sections, show that bank-level filtering units are so effective that they reduce time spent on filtering to a negligible proportion of execution time and minimize the amount of data that subsequently needs to be fetched by the CPU. This approach is so effective that a more aggressive PIM approach, such as subarray-level PIM, rarely provides meaningful additional end-to-end performance benefit—with bank-level PIM, the filtering step is already reduced to such a small portion of the overall execution time that further hardware cost to achieve greater speedup is not worth the additional hardware cost. But in comparison to channel- or rank-level processing, the bank-level approach provides significant speedup, with tiny hardware cost.

Once the filtering kernel has produced its output bitmap, the rest of the query is processed on the CPU. The necessary fields from only the selected records, as indicated by the bitmap, are fetched to the CPU.

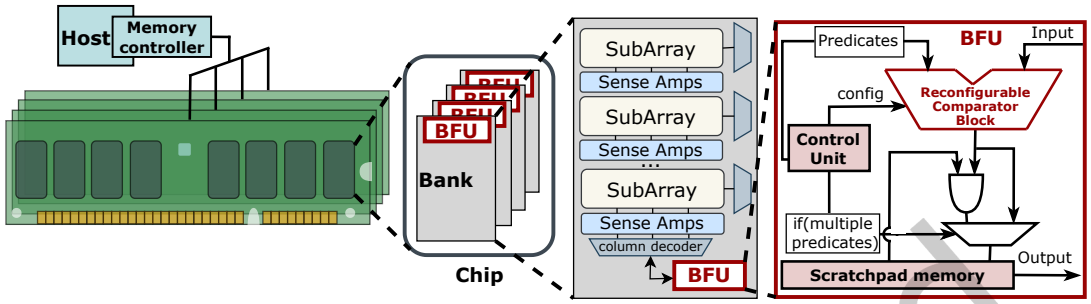


Fig. 3. System with Membrane Bank-level Filtering Unit

4 PIM Architectures

4.1 Bank-level Filtering Unit (BFU)

Our proposed Bank-level Filtering Unit (BFU) only needs to support comparisons for the filtering step. The BFU is placed at each bank interface, which fetches 64 bits in a burst from the subarray row buffer. The BFU processes data every tCCD_L in All-Bank mode, like Aquabolt [37]. The breakdown of BFU is presented in Figure 3, and its components are described below.

The BFU's **Reconfigurable Comparator Block (RCB)** can support both equality check (database_value==a) and range check (a<database_value<b) on integer and floating point values. Each comparison within BFU produces a result bit that is placed into a bitmap stored in the BFU's bitmap buffer, which is 64 bits. When performing a sequence of multiple filtering operations, the RCB reads the value of the bitmap for the current position and ANDs this with the result from the new comparison operation. This way, the bitmap accumulates the final Boolean result for an entire query. Once the 64-bit output buffer is filled, it is written back. The RCB does not support processing string or regex operations, as these are infrequent and more costly for PIM implementation. In any case, strings would be dictionary-encoded.

Modern databases store columnar data in a bit-packed format to decrease memory usage, so that fields with a small range, such as zip codes, do not waste space. Our comparison unit supports any bit length from 2 to 64 bits, with smaller bit widths processed in SIMD fashion. Configuration bits indicate the data type to be processed. Thus, before processing a database column, the RCBs are configured to the specified bit length, programmed with the predicate values to compare against, and then the filtering operation begins to produce the desired resultant bitmap. The **Control Unit** orchestrates fetching data from the DRAM bank and performing the comparison at the desired bit lengths.

4.2 Subarray-level Filtering

To explore subarray-level parallelism, we adapt Fulcrum architecture [38], which places a PE at the edge of the subarray, and uses Walkers (row-wide buffers) and column-select logic to move input operands out to the PE, and move output values to the appropriate location in the output Walker. Due to the open-bitline architecture, we can have at most one PE for every two subarrays. The full Fulcrum architecture supports arithmetic, etc., while filtering only needs comparison, and it only needs two Walkers to read the input column values and capture the resulting bitmap values. Our subarray-level PE is similar to the BFU, with a Walker used to hold the bitmap. The second Walker means the area overhead per filtering unit is significantly higher than the bank-level approach. For our chosen DRAM configuration (Table 2), CACTI [16] indicates 16 subarrays per bank with each subarray containing 4096 rows. For 16 subarrays, the maximum subarray-level parallelism (SALP) is at most 8, because each PE is shared by two subarrays. For smaller degrees of SALP, data may need to be moved from a subarray that does not have access to a PE to one that does, using the LISA technique [14].

As shown in Figure 6, subarray-level PIM provides minimal extra performance compared to the bank-level approach, at the cost of higher area, so we do not consider further design optimizations.

4.3 Rank-level and Channel-level Filtering

Placing computation outside the DRAM on the DIMM module or in the memory controller avoids changes to the DRAM but also gives up the higher internal parallelism of the DRAM. We explore the rank-level filtering (similar to [32]) by placing a filtering unit in a buffer chip on the DIMM circuit board. This rank-level filtering unit can process an entire 64-bit DRAM read burst in one cycle, and is an upper bound for filtering throughput outside the DRAM chips if we maintain a standard-width DRAM interface. For the channel-level filtering, we place a similar unit in the memory controller. This channel-level filtering unit provides a rough approximation of what the Intel Analytics Accelerator (IAA) [29] can achieve, by offloading filtering from the cores and avoiding cache pollution. We model the channel-level filtering in this way, like the rank-level filter unit, for better comparison with the rank-level approach, and because we were not able to find specific implementation details of the IAA. The only significant difference between our channel- and rank-level filtering is that the rank-level approach has one unit per rank, thus achieving rank-level parallelism.

As our results show, the speedup at the bank level, compared to rank- and channel-level, is substantial (proportional to the number of banks), so we do not consider further optimizations for rank- and channel-level processing.

4.4 System Integration

Following the Aquabolt approach [37], which maintains compatibility with existing DRAM interfaces, Membrane supports Single-Bank (SB) mode (normal read/write) and All-Bank (AB) mode for PIM. In AB Mode, a DRAM READ command to a specific address reads data at the address into the local BFU and performs a PIM computation (i.e., comparison). In this mode, the bank and bank-groups bits in a given memory address are ignored, and data at the same column position across all the banks is read into the local BFUs. As with Aquabolt, Membrane uses MRS (Mode Register Switch) and PIMCONF (PIM Configuration) registers to control the functionality of PIM processing elements. MRS is used to transition between the normal mode (SB mode) and PIM-capable mode (AB Mode). PIMCONF registers are used to program the PIM processing elements with instructions. In our case, we use the PIMCONF registers to program the BFU with the values to be compared against during the predicate operations and set the processing bitwidth.

Cacheline De-interleaving Unit (DU). A cache line is read or written in 64-bit chunks, and a single 64-bit chunk of data is usually striped across multiple chips within a single DDR rank. Traditionally, DDR comes in x4, x8, or x16 configurations, in which each xN chip in the rank contributes 4, 8, or 16 bits to a 64-bit DRAM access. This striping across multiple chips is problematic for PIM when processing operands greater than the 4-, 8-, or 16-bit width, because different bytes of an operand are spread across multiple chips, preventing even simple comparisons. In order to support the PIM-compatible data layout, we add a cache-line-wide buffer that stores and swizzles the bits in a cache line before writing to DRAM or reading from DRAM, such that the entire operand resides in one single DRAM chip. Based on our discussions with two major CPU vendors, the overhead associated with routing incoming bytes to appropriate locations in this buffer is likely to have negligible impact on area and no impact on timing.

PIM Pages. When running in AB Mode, each DRAM command triggers a READ and PIM operation across a single database column in the same row across all banks. A *PIM page* is the enforced minimum allocation unit for PIM, and is a multiple of the native operating system superpage size. The PIM page size will depend on the system's configuration, so the PIM page fills at least one entire system-wide DRAM row, i.e. spanning this row "position" across all banks, ranks, and channels. For a large memory system, this will require multiple

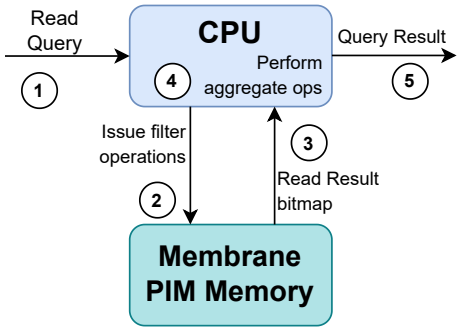


Fig. 4. Query Execution Flow with Membrane

contiguous superpages, e.g., for an 8-channel, four ranks/channel system with DDR4_8Gb_x8_2933 DRAM chip configuration taken from [40], a PIM page is 4 MByte, requiring two 2 MByte superpages on an x86-64 system. For a smaller system, a single superpage will suffice, and it might occupy several logical rows. This means that when building a system to use Membrane memory, the memory system should configure memory channels in powers of 2. If the data does not fully occupy a PIM page, padding should be applied, and any results associated with the padded data must be disregarded when transferring the results back to the Host CPU.

Using superpages allows us to allocate PIM data with permissions enabling PIM. The operating system must support a new version of malloc, filter_malloc(), that gives the owner permission to issue PIM commands to this region of memory. A filter_malloc is needed for each input PIM page as well as each output PIM page, for storing the output bitmap. PIM permission must be noted in the page table and requires one extra bit in the TLB. The system must also support virtual-to-physical translation and permission checking at the granularity of PIM pages; more on this in the next subsection. No other changes are required to the operating system or MMU. Leveraging the superpage feature benefits from the reduced TLB lookups provided by superpages. Note also that PIM pages do not need to require any particular placement in memory or relative to each other.

Query Processing and Additional System/Hardware Support Membrane requires several new CPU instructions to control PIM operation, as noted below. The description below is specific to bank-level PIM, but can easily be adapted to subarray-level, etc. The overall execution flow is shown in Figure 4. The software performs PIM filtering on a column by operating on one PIM page at a time. We envision this being implemented in a PIM user-level filter library that any database engine can incorporate. Only one thread in the database should perform PIM.

The library first issues a pim_begin() system call. This allows the OS or hypervisor to manage access to PIM mode and return an error if the calling process is not allowed to use PIM. It can also support fair sharing for PIM access, etc. Although Aquabolt [37] allows user-level access to PIM mode, we suggest that access be mediated by the OS. The pim_begin() system call writes into the MRS register to drain the memory controller queues and switch all channels into All Bank mode. This also blocks regular load-store access from other threads/cores. When the system call returns, the user-level library programs the BFUs with the predicate values and operation type using a PIM_CONF instruction that writes to the PIMCONF address, which is broadcast to all banks' BFUs.

Now filtering can begin using a sequence of PIM_FILTER instructions, each specifying one PIM input page, one PIM output page, and an offset within the output page (because input values may be up to 64 bits but the corresponding output value is just a single bit, so processing an entire input PIM page only fills up a small portion of the output page). These addresses are translated through the TLB to obtain the physical address for the PIM page and verify PIM permission, and these physical addresses are sent to all memory controllers. For each PIM_FILTER instruction, the memory controllers process their portion of the PIM page by sending the

appropriate sequence of PIM DRAM commands to activate the target DRAM row and sequence through this row's DRAM columns. The latency for each of these DRAM commands is deterministic, so the memory controller knows how long to wait before initiating a subsequent command. A PIM_FILTER instruction is issued for each PIM page needed to complete the filtering required by the column. Once the filtering of the database column is complete, a new column can be processed. When PIM computation is complete, the library issues a pim_end() system call, which reverts the memory to standard operation. Note that we do not attempt to offload the PIM computation from the core, because AB mode blocks the entire memory system anyway, and the calling thread is waiting on the PIM results. In summary, PIM is not completely transparent to the software. The database application needs to allocate the data in memory using pim_malloc(), and use the PIM-enabled filter library.

Filtering is idempotent, so if a non-maskable interrupt occurs that requires DRAM access, for simplicity, the filter operation can be aborted and restarted later. Any prior partial bitmap results will be recomputed. This avoids the need to preserve partial PIM state.

For other interrupts, the OS can wait until a PIM page has been completed, then transition out of PIM mode if needed. If the process performing PIM computation needs to be suspended, the OS only needs to remember to re-initiate PIM mode when this process resumes. The process's progress in the filtering task is remembered in its user state.

Our modeling based on the Samsung Aquabolt PIM Simulator [46] indicates that if any incoming regular read request is allowed to immediately interrupt PIM processing, and we assume a worst-case scenario in which a constant stream of reads causes a mode switch after processing every individual GDL chunk, the bank-level filtering latency increases by 23.9x. Despite this, it remains 5.4× faster than the CPU-only filtering approach. Under these conditions, the geometric mean speedup for SSB decreases from 5.9× to 4.8× (Sec. 5), with the most significant impact observed in queries that initially benefited the most from PIM filtering. For example, Q3.4, which achieved a speedup of 25.9× without interruptions, sees its speedup drop to 9.9× under frequent switching overheads. However, suppose PIM filtering is allowed to proceed for at least 50 ns without interruption (i.e., a total memory blocking interval of 200 ns, comprising 150 ns for RT mode switching plus 50 ns for filtering). In that case, the geomean speedup only drops slightly from 5.9× to 5.7×, and Q3.4 sees a more modest decrease from 25.9× to 21.9× (see detailed results in Section 5).

No changes are needed to support multi-tenancy, since each VM will have its own memory allocations. The hypervisor will need to support PIM mode. Switching between separate PIM-enabled database instances does not incur the cost of the mode switch, and fairness could be achieved by switching after processing a fixed number of PIM pages.

Once all filtering is completed for the query, the database software now reads the bitmaps from the main memory, and accesses any needed fields for records that have been selected. The aggregate operations (sorting, group by, average or as such) to produce the final result.

Privilege Level and PIM Instruction Semantics. Following the Aquabolt design philosophy [37], our PIM operations are exposed as user-level instructions that translate into sequences of standard DRAM READ/WRITE commands at the memory controller. This avoids reliance on privileged commands, which require kernel intervention and introduce substantial overhead due to system calls and context switching. Instead, we reserve a small memory region, PIM CONF, that maps to two configuration registers—the AB Mode Register (ABMR) and the SB Mode Register (SBMR). Transitions between Single-Bank (SB) and All-Bank (AB) modes are performed entirely in user space by issuing carefully crafted sequences of standard ACT/PRE commands to specific addresses in the PIM CONF memory region. This approach maintains full compatibility with JEDEC-compliant DRAM controllers and allows unprivileged processes to invoke AB-mode PIM operations without elevated privilege levels.

Trap Behavior, Return Paths, and Execution Flow. PIM instructions execute without generating traps or interrupts. At the CPU level, a PIM instruction simply enqueues a DRAM command sequence targeting the

PIM memory region. Once the program reaches a designated ordering point (e.g., a memory fence), the core guarantees that all previously issued PIM commands are visible to the memory controller before transitioning to PIM execution. The PIM operations proceed entirely within DRAM, and their results are written back to architecturally visible memory locations. After the PIM sequence completes, the CPU resumes normal execution and reads the results via standard load instructions. No asynchronous events, interrupts, or special return paths are required; the instruction flow remains fully synchronous from the CPU's perspective.

Ordering Semantics and Interaction with Out-of-Order Cores. To guarantee correctness, we place ordering constraints only at the boundaries of PIM execution phases. Before transitioning into AB mode, the host issues a memory fence to ensure that all in-flight CPU memory requests have completed. Similarly, after the final PIM command, a second fence ensures that all AB-mode operations have finished before the CPU resumes normal execution.

Crucially, these fences do not constrain the ordering of instructions within the PIM region. Once PIM commands are dispatched to the memory controller, the DRAM subsystem dictates the execution semantics. In AB mode, the DRAM ranks and banks ignore the bank address bits and broadcast each command to all banks simultaneously, effectively collapsing the memory array into a single logical bank. Thus, all PIM operations in AB mode are inherently serialized, and the memory controller cannot exploit bank-level concurrency or overlap independent operations. Because AB-mode execution is already globally serialized at the DRAM level, enforcing strict ordering at the CPU boundary does not result in any additional performance penalty. Any potential parallelism that an out-of-order CPU might provide is already moot under AB-mode constraints.

Ordering Fences. In conventional DRAM systems, excessive fencing harms performance by suppressing bank-level and memory-level parallelism. In AB mode, however, this parallelism is restructured rather than removed: the hardware executes the same row/column command across all banks in lock-step, effectively transforming independent bank-level parallelism into coordinated all-bank parallelism. Because each PIM command occupies the entire DRAM subsystem during its execution window, the hardware already prevents overlapping or out-of-order activity across banks. CPU-level ordering fences therefore simply align the processor's view of memory with the serialization already imposed by AB-mode PIM execution. As a result, fences at mode-transition boundaries introduce negligible overhead beyond the inherent lock-step operation of AB mode.

5 Experimental Methodology

DRAM simulation and host system. In Membrane, filters are executed in PIM, and the remaining work of the query is executed on the host CPU, taking the bitmap as input. To model the PIM portion, we use DRAMSim3 [40], a cycle-accurate DRAM simulator. When a PIM page is activated, the entire page is brought into the row buffers across all the banks and all the ranks, and then each bank processes its portion in 64-bit chunks. The precharge, row activation, and reads are modeled in DRAMsim3 to obtain the time required to filter an entire PIM page in AB mode. To evaluate the host CPU portion, we use DuckDB, a state-of-the-art OLAP database system [50]. DuckDB is extensively optimized, outperforming more established systems by orders of magnitude in many cases, and is widely used as a baseline in database research [20, 23, 30, 31]. The total end-to-end time spent processing a query is the sum of time spent on PIM filters (simulated) and the rest of the query in DuckDB (real-world execution). Note that, to avoid iterating through sparse bitmaps, we modify DuckDB to leverage CPU instructions that count the number of trailing zeroes in a word.

Workload. To evaluate the many dimensions of the DRAM-PIM filtering design space, we use two established OLAP benchmarks: TPC-H [2] and the Star Schema Benchmark (SSB) [47]. TPC-H is a widely used OLAP benchmark designed to comprehensively assess the performance of OLAP systems. The TPC-H database consists of a large, central `lineitem` table emulating the items ordered from a business. Dimension tables store information about parts, suppliers, customers, and locations. The benchmark consists of 22 queries. To focus our evaluation,

Table 2. Configuration Details.

Property	Value
Baseline System	Intel Xeon Silver 4314 @ 2.40 GHz
Total Cores / Main Memory	16 Cores (32 threads) / 128 GB (8-ch DDR4-3200)
PIM Config.	DDR4_8Gb_x8_3200; 8-channel, 4-rank/ch, 4-bank-group; 4 banks per bank-group, 16 subarrays/bank

Table 3. Database size in GB for varying scale factor (SF).

	SF-10	SF-20	SF-50	SF-100
SSB	6.8	8.6	14.4	23.5
TPC-H	7.8	10.8	21.7	39.3

we use a subset of 8 queries that have been used in prior work focused on evaluating filter performance [58]. We report the geometric mean of this subset to summarize our results. While SSB is based on TPC-H, it includes notable distinctions that aim to improve its accuracy and coverage as a benchmark. SSB combines the `lineitem` and `orders` tables, a standard technique used to avoid unnecessary joins [33]. It also drops tables and columns that are unlikely to be present in an OLAP database, such as string comments and shipping instructions. The query suite consists of 4 “flights", each of which models a common OLAP pattern. Within each query flight, there are several queries with varying selectivity (the number of rows that contribute to the result of each query). The database itself consists of one large fact table and four smaller dimension tables.

Membrane Circuit Evaluation. We implement Membrane’s Bank-level Filtering Unit in RTL and use Synopsis DC Compiler in 14 nm to evaluate its delay, power, and area. We use scaling factors from Stillmaker et al. [57] to scale down the results to 22nm. Each BFU occupies $0.001mm^2$ area, which is negligible, and has a path delay of $0.45ns$, which easily fits within the column-to-column access time. The power for one BFU is $118.7uW$, which, when aggregated across all banks within a rank, is 2.3% more than the regular DRAM operation. However, AB mode operates all banks at once, which increases peak power. Another PIM architecture [28] that leverages the AB mode observes that the peak power increases by 4x when operating in this mode. By engaging every bank in parallel, AB mode incurs a four-times higher power draw than SB-mode DRAM when operating at peak read bandwidth [28]. If the DIMM memory slots cannot supply enough power, an alternative way is to integrate external power delivery mechanisms—such as NVDIMMs [45], an approach adopted by other PIM implementations such as [6].

Our evaluations consider these increased power requirements. Overall, we observe a 3.6/3.1x geometric-mean energy efficiency gain over a baseline system without Membrane for SSB/TPCH benchmarks, respectively.

Energy Consumption Analysis. We estimate the power consumption of CPU while performing filter and non filter kernels based on CPU usage using the methodology in [10]. The overall energy consumption is obtained by integrating CPU power, DRAM power (obtained from DRAMsim3), and BFU power (obtained from the RTL analysis above) over the time spent on the filter and non-filter kernels of end-to-end query execution. In AB mode, the extra power is included for the duration of PIM execution. The relative energy efficiency highly correlates with the end-to-end execution time of the queries. ***We observe higher energy efficiency (~20x) with more selective queries such as Q3.3, Q3.4, Q19.***

Denormalization. To improve PIM amenability and fully exploit Membrane’s capabilities, we explore the use of denormalization. As introduced in Section 2.1, denormalization involves joining tables as a database is loaded. Commonly used to reduce query complexity and improve performance, denormalization replaces joins with

Table 4. Levels of Schema Denormalization.

Denorm. Level	Description
D1	No Denormalization (Plain Schema)
D2	Denormalizing columns used in WHERE clause only
D3	D2 + Columns used for aggregate operations
D4	Full Schema Denormalization

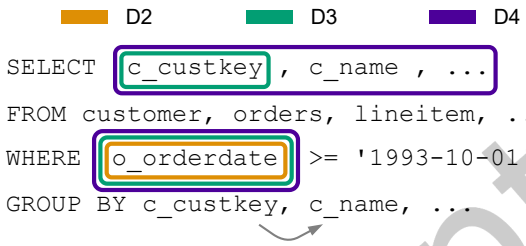


Fig. 5. Illustrating which columns are denormalized in each level with TPC-H Q10. In D3, c_name is not denormalized because it is functionally determined by c_custkey.

filters. However, denormalization requires extra space to store denormalized data. The choice of which columns to denormalize presents a tradeoff between PIM amenability and space overhead, as shown in Table 4.

Prior work, such as [44], has explored automated approaches like cross-training to identify columns suitable for denormalization. In contrast, our study aims to showcase the potential benefits of partial denormalization, guided by the queries present in the selected benchmarks. In practice, decisions regarding which columns to denormalize are typically made by schema designers and database optimizers, taking into account application requirements, historical query patterns, and broader usage characteristics.

At one extreme (D1), we can avoid denormalization, which incurs no space overhead but limits the speedup. At the other extreme (D4), we can denormalize all columns, which maximizes PIM amenability at the expense of considerable space overhead. As reported in prior work, full denormalization can result in a space blowup of over 10x [42].

We propose two denormalization levels, D2 and D3, which fall between the two extremes, offering a better balance between PIM amenability and space overhead. Inspired by WideTable [42], we use static analysis of the workload to choose a subset of columns to denormalize. In addition, we use dictionary encoding and bitpacking compression in all our experiments to reduce space overhead, which is particularly beneficial for denormalization. These techniques do not affect PIM amenability.

In D2, we denormalize a column if it appears in the `WHERE` clause of any query in the workload. Recall from Section 2.1 that in D1, rows of interest are selected through a combination of filters and joins. D2 replaces these joins with filters.

D3 is motivated by the observation that a significant portion of query time is spent on joins even after denormalizing columns that appear in the `WHERE` clause, as shown by the D2 breakdown in Figure 8b. For certain queries, joins are used to retrieve columns that are not in the `WHERE` clause but are still needed by the query. In D3, we reduce the impact of these joins by denormalizing a column if it appears in the `WHERE` clause or the `SELECT` clause of any query in the workload. To reduce space, D3 involves a notable exception: we do not denormalize dimension table columns that only appear in the `SELECT` clause and are functionally determined by another column in the `GROUP BY` clause. The exception is based on the observation that group-by aggregation and limit operations often reduce the number of rows down to the order of tens to hundreds. After these operations have

Table 5. Single-column filter latency and processing throughput across different DRAM heirarchy

PIM Arch.	Chnl	Rank	Bank	SALP-2	SALP-4	SALP-8
Time (ms)	32.4	8.46	0.28	0.08	0.04	0.02
Throughput (GBps)	37.03	141.4	4285.71	15000	30000	60000

been completed, an inexpensive join can be used to retrieve the functionally dependent columns and produce the result. As shown in Figure 5, D3 avoids denormalizing `c_name` and other large columns from the `customer` table.

6 Results

6.1 Filter Performance Across DRAM Hierarchy

We previously explained why the filter kernel is the most suitable candidate for PIM acceleration and suggested that the bank level is the best choice for adding PIM computation for filtering. Now we substantiate this claim by briefly evaluating the benefits of placing PIM filtering at different levels of the DRAM hierarchy.

In order to assess the benefit of filtering at different levels of the DRAM hierarchy, we constructed a microbenchmark that performs a simple predicate (a<input_value<b) evaluation on one column of the Star Schema Benchmark's (SSB) fact table. Each fact table column at scale factor-100 contains 600,038,146 elements; for this microbenchmark, the values are 16 bits each (for a total of 1.12 GB).

Table 5 shows the latency for our microbenchmark with different forms of near/in-memory processing. While 8-way subarray-level parallelism is able to achieve 14x speedup over the bank-level approach on our microbenchmark, when considering geometric-mean end-to-end performance of the SSB and TPC-H suites, as shown in Figure 6 along with area overhead, the speedup advantage with SALP over baseline CPU drops to 1.1x speedup (SALP-8 vs Bank-Level), which does not appear to justify the much higher area cost.

Comparing between rank-level and bank-level, we observe that rank-level PIM does not have any area overhead inside the DRAM chips, but it is 29.4x slower than the bank level approach in the microbenchmark. However, when considering the geometric-mean end-to-end performance of the SSB/TPC-H suites, the bank-level solution offers 1.89x/1.59x speedup over rank-level with 4 ranks/channel.

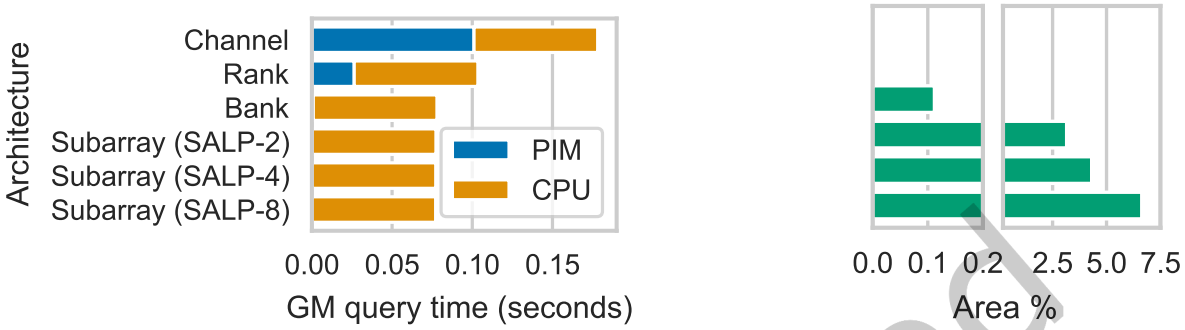
Filtering could also be performed in the memory controller or some other unit in the CPU, as in the Intel Analytics Accelerator [29], which offloads this data-intensive task from the cores and avoids cache pollution, but gives up the rank-level parallelism of the rank-level solution. Bank-level offers 3.7x/3.15x speedup (SSB/TPCH) over this channel-level solution.

Based on these findings, we conclude that the bank is the best level of the DRAM hierarchy in which to implement filtering, with only 0.1% area overhead relative to the baseline DRAM chip area.

6.2 Partial denormalization enables more extensive acceleration

We evaluated the overall performance of Membrane bank-level PIM's performance with SSB and TPC-H benchmarks against the baseline system configuration (Table 2). Speedups directly correlate with query selectivity. We observe that with the accelerated PIM filters, we obtain end-to-end geo-mean query speedup of 5.92x/6.38x in SSB/TPC-H while using the D3 schema, but only 1.2x and 1.3x for SSB and TPC-H if denormalization is not used (D1).

To better understand the benefits of denormalization, in Figure 8, we show the average percentage of time spent in each operator for SSB and TPC-H without PIM acceleration. For denormalization level D1 (the standard schema), scans (both with and without filtering) account for 60% and 51% of query time in SSB and TPC-H. As shown in Figure 7, for denormalization level D1, Membrane achieves over 2x speedup for SSB Q1.2 and Q1.3 and TPC-H Q6 and Q14, which are dominated by filtering. Unfortunately, because scans with filters account for only 22% of overall SSB query time and 46% of overall TPC-H query time, Amdahl's law limits the overall potential



(a) Query Time (b) Area Overhead

Fig. 6. Geometric mean SSB query time and area overhead (relative to cell area) for varying PIM architectures.

speedup to approximately 1.3x and 1.9x. However, Figure 8 shows that a substantial portion of time in D1 is also spent on joins, which dominate after D1 filtering is accelerated by PIM.

At the expense of 17% and 9% extra space, as shown in Figure 9, D2 yields geometric mean query speedups of 5.9x and 4.8x for SSB and TPC-H. Denormalization without Membrane acceleration also improves performance, but to a much lesser extent. D3 further increases the portion of query time that Membrane can accelerate. At the expense of 3% extra space, D3 achieves a geometric mean query speedup of 6.4x for TPC-H. For SSB, D2 and D3 happen to be equivalent. Figure 8 shows that with D3, joins have been nearly eliminated and converted to filters, and most of the execution time has been converted to filters.

6.3 Speedup tends to increase as database size increases

We now investigate the effect of database size on query speedup. Recall that the number of rows in each table is proportional to the scale factor, with the exception of the `part` table in SSB, which scales logarithmically. As shown in Table 3, database size is roughly proportional to scale factor.

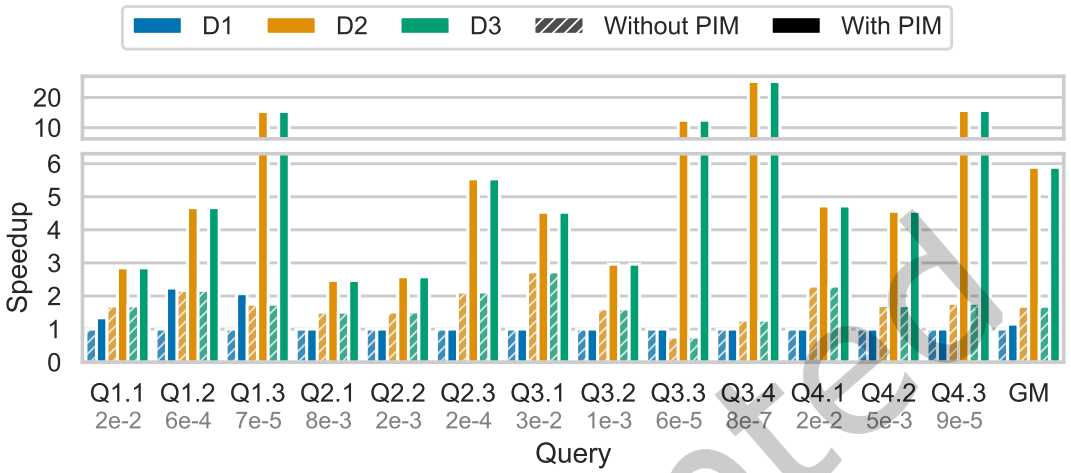
Varying the scale factor from 10 to 100, we evaluate Membrane's performance for denormalization levels D1-3, shown in Figure 10. We observe that query speedup tends to increase as the database size increases. At scale factor 10 and denormalization level D3, the geometric mean query speedups are 4.4x and 5.0x for SSB and TPC-H. At scale factor 100, the speedups increase to 5.9x and 6.4x. Database size has little effect on query speedup without PIM.

To explain the effect of database size of query speedup, we measured average CPU usage for each configuration. We note that the CPU usage reported here excludes the period spent waiting for PIM filtering to complete. Results are shown in Figure 10. At smaller scale factors, CPU usage with PIM is significantly lower than CPU usage without PIM.

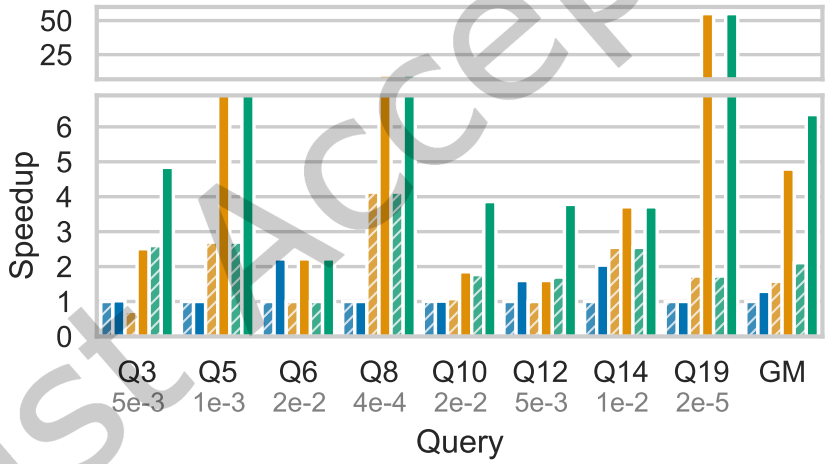
During query processing, database systems typically incur overheads for parsing, planning, optimization, and scheduling. Although DuckDB is extensively optimized, at small scale factors, the CPU has very little data left to process after PIM filtering, so these overheads play an outsized role.

6.4 Speedup tends to increase as PIM selectivity decreases

We now explore the impact of PIM selectivity on query speedup. We define PIM selectivity as the fraction of rows returned by PIM after filtering, or equivalently, the fraction of set bits in the bitmap. A given query's PIM selectivity may depend on the denormalization level. For example, TPC-H Q3 has a PIM selectivity of about 0.54 for D1 and 0.005 for D2.



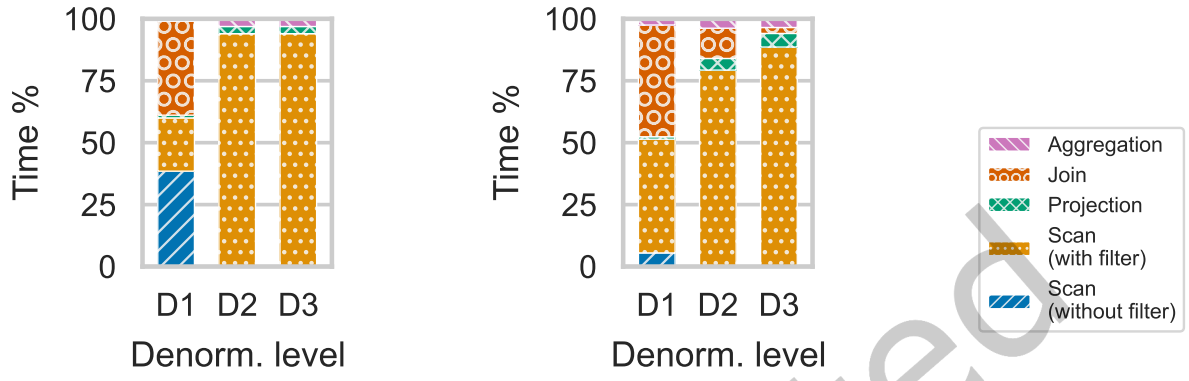
(a) SSB



(b) TPC-H

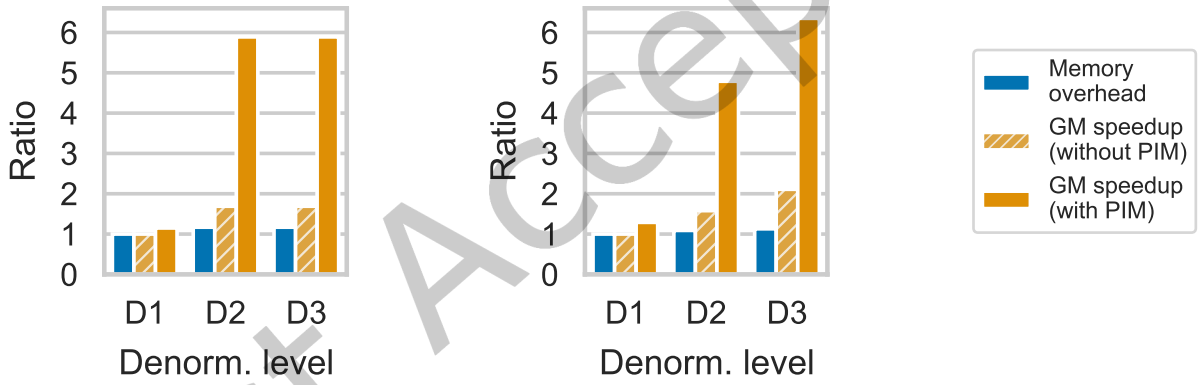
Fig. 7. Query speedup for varying denormalization level (relative to D1 without PIM). Query selectivity is shown at the bottom.

In Figure 11, we show query speedup versus PIM selectivity for combined SSB and TPC-H. Each point in the plot is an individual query. We observe that query speedup tends to increase as PIM selectivity decreases. All queries with PIM selectivity less than 10^{-4} are at least 10x faster in Membrane. In contrast, among queries with PIM selectivity greater than 0.1, the maximum query speedup is 1.3x. Fortunately, analytical queries usually include filters with low selectivity, which can be accelerated in Membrane after partial denormalization. For D3, the minimum and maximum PIM selectivities are 7.6×10^{-7} and 0.034, respectively, and the minimum and maximum query speedups are 2.2x and 57x, respectively.



(a) SSB (b) TPC-H

Fig. 8. Average operator time percentage for varying denormalization level (without PIM).



(a) SSB (b) TPC-H

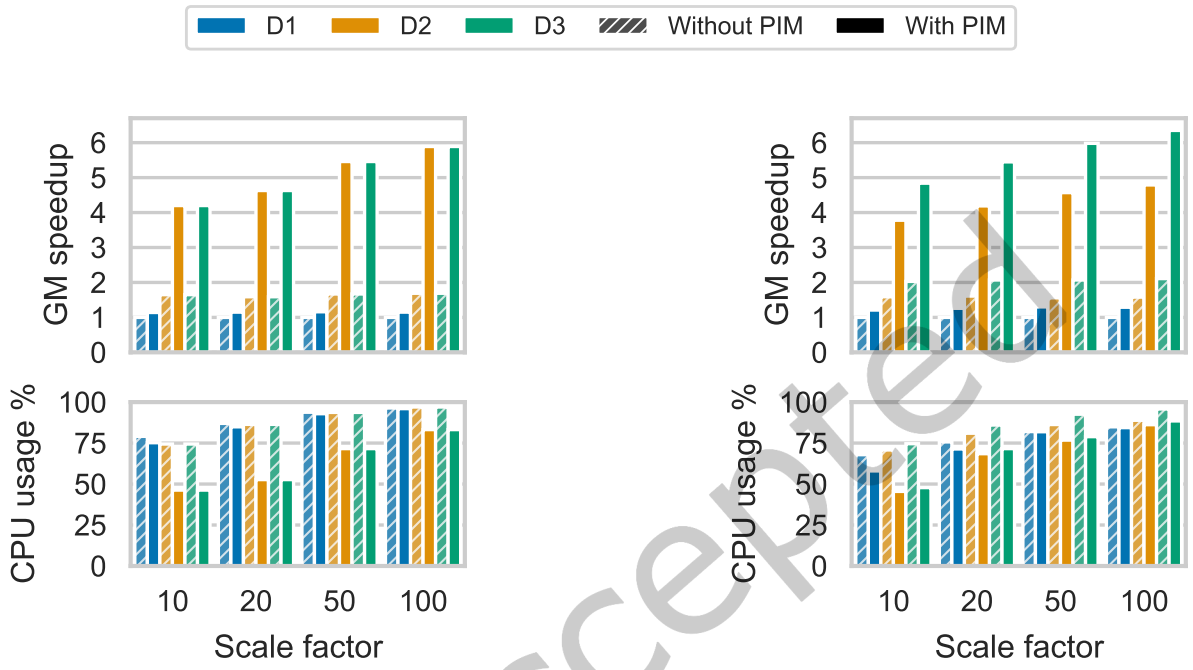
Fig. 9. Memory overhead and geometric mean query speedup for varying denormalization level (relative to D1 without PIM).

7 Related Work

Prior works in the database field, such as BitWeaving [41], exploited the "intra-cycle"/bit-level parallelism of processors to accelerate the scan and filtering kernels. SIMD-scan [59] aimed to perform the same by utilizing on-chip vector processing units with SSE instructions.

Processing In Storage Solutions. With database machines [12], there were attempts in the 1970s and 1980s to push query computation closer to where the data resided—at that time, spinning disks. However, these efforts were abandoned because the resulting custom storage package was expensive to manufacture, in stark contrast to the commodity microprocessors that were experiencing exponential performance gains at the time. However, now, with the slowing of Moore's Law, there is a need to revisit ideas for specialization in today's context. Pinatubo [39] and SmartSSD [19] are examples of other works that have proposed pushing query processing into the storage device. These designs, however, are limited by the storage I/O interface and suffer from higher latency and lower degrees of parallelism, and do not serve the needs of markets using in-memory databases.

DRAM-Based PIM Designs. Several prior works, such as Ambit [54] and SIMDRAM [26], propose a triple-row activation design to perform logical operations at the subarray-level that could be leveraged for processing



(a) SSB (b) TPC-H

Fig. 10. Geometric mean query speedup (relative to D1 without PIM, same scale factor) and average CPU usage for varying scale factor and denormalization level.

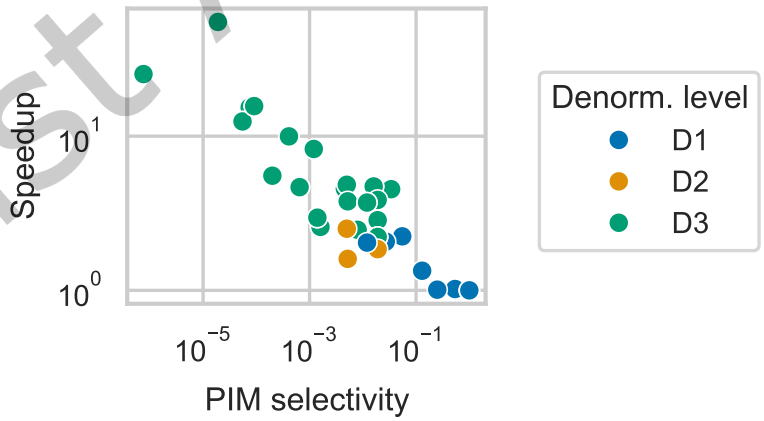


Fig. 11. Query speedup for varying PIM selectivity and denormalization level (relative to D1 without PIM).

OLAP queries, but these approaches require more significant changes to the DRAM, in particular support for multiple concurrent row activations per bit-level operation. JAFFAR [62] is a DIMM-level design that focuses on the filter operation by operating on the I/O buffer present on each DIMM. This approach is similar to our rank-level. The Reconfigurable Vector Unit [52] proposes to implement vector processing units at a vault-level in

an HMC design. Polynesia [13] accelerates the analytical portion of HTAP database workloads using vault-level processing elements on HMC. Our approach would also extend to HMC or HBM, but in-memory databases benefit from the greater capacity scaling of conventional DIMMs. Most prior works have explored offloading scans to processing in/near memory, such as rank-level processing, e.g., with Samsung's AX-DIMM [36], but fail to evaluate end-to-end query processing pipelines on popular benchmark suites such as SSB and TPCH. Membrane differs from most of these works in that it thoroughly explores the design space for conventional DIMM memory and cooperatively processes the entire query together with the host, rather than in a production DBMS, DuckDB, instead of offloading the entire query processing to PIM hardware or only evaluating kernels.

Alternative Architectures. Prior works such as [8] accelerated the filtering step on the GPU but omitted the data-retrieval portion and subsequent postprocessing, which we have shown will often consume a large portion of query processing time. Ibex [60] and [63] implemented query processing on FPGAs. However, GPUs and FPGAs suffer from the limited scalability of onboard memory compared to the main memory addressable by the CPU. Papaphilippou and Luk [48] provides a comprehensive survey of works investigating acceleration of database systems using FPGAs and arrives at similar conclusions.

We implemented an optimized version of the filter kernel on an Alveo U280 FPGA to leverage the onboard HBM memory to execute the filter microkernel described earlier in Section 6.1. We observe that Membrane outperforms this FPGA setup by at least 27.46x, including the cost of data transfers to and from the FPGA.

CPUs, discrete GPUs, and similar processing units can utilize Membrane bank-level PIM units for filter and transfer intermediate results efficiently back to the hosts.

8 Conclusions

In-memory analytics can be accelerated by offloading the filter kernels to PIM processing units. In this work, we observe that denormalization methods make these workloads significantly more amenable to PIM filtering, albeit by incurring extra memory overheads. We evaluated different levels of denormalization that provide a tradeoff between increased memory consumption and improved performance. We thoroughly explored the DRAM design space to conclude that bank-level offers high performance with minimal area overhead and power usage. Membrane's bank-level PIM can outperform the CPU baselines by 5.9-6.3x and have a memory overhead of 9-17%, depending on the different denormalization levels across both TPCH/SSB benchmarks.

9 Acknowledgements

This work is funded in part by the National Science Foundation (NSF) under collaborative awards CCF-2312739, CCF-2312740, CCF-2312741, and CCF-2407690 as well as PRISM and ACE, two of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program, sponsored by DARPA.

References

[1] [n. d.]. Indexes. https://duckdb.org/docs/sql/indexes.html. Accessed: 2024-04-18.
[2] [n. d.]. TPC-H Benchmark Specification. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.1.pdf
[3] 2021. Business Intelligence and Analytics Market. https://www.emergenresearch.com/request-sample/467.
[4] 2022. In-Memory Database Market. https://www.alliedmarketresearch.com/in-memory-database-market-A31497.
[5] Daniel Abadi, Anastasia Ailamaki, David Andersen, Peter Bailis, Magdalena Balazinska, Philip A. Bernstein, Peter Boncz, Surajit Chaudhuri, Alvin Cheung, Anhai Doan, Luna Dong, Michael J. Franklin, Juliana Freire, Alon Halevy, Joseph M. Hellerstein, Stratos Idreos, Donald Kossmann, Tim Kraska, Sailesh Krishnamurthy, Volker Markl, Sergey Melnik, Tova Milo, C. Mohan, Thomas Neumann, Beng Chin Ooi, Fatma Ozcan, Jignesh Patel, Andrew Pavlo, Raluca Popa, Raghu Ramakrishnan, Christopher Re, Michael Stonebraker, and Dan Suciu. 2022. The Seattle report on database research. *Commun. ACM* 65, 8 (Aug. 2022), 72–79. doi:10.1145/3524284
[6] Mohammad Alian, Seung Won Min, Hadi Asgharimoghaddam, Ashutosh Dhar, Dong Kai Wang, Thomas Roewer, Adam McPadden, Oliver O'Halloran, Deming Chen, Jinjun Xiong, Daehoon Kim, Wen-mei Hwu, and Nam Sung Kim. 2018. Application-Transparent Near-Memory Processing Architecture with Memory Channel Network. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 802–814. doi:10.1109/MICRO.2018.00070

[7] Johnathan Alsop, Shaizeen Aga, Mohamed Ibrahim, Mahzabeen Islam, Andrew Mccrabb, and Nuwan Jayasena. 2024. Inclusive-PIM: Hardware-Software Co-design for Broad Acceleration on Commercial PIM Architectures. (2024). arXiv:2309.07984 [cs.AR] doi:10.48550/arXiv.2309.07984

[8] Peter Bakkum and Kevin Skadron. 2010. Accelerating SQL Database Operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*. 94–103. doi:10.1145/1735688.1735706

[9] 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 *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 362–373. doi:10.1109/ICDE.2013.6544839

[10] Robert Basmadjian, Nasir Ali, Florian Niedermeier, Hermann de Meer, and Giovanni Giuliani. 2011. A methodology to predict the power consumption of servers in data centres. In *Proceedings of the 2nd International Conference on Energy-Efficient Computing and Networking* (New York, New York, USA) *(e-Energy '11)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/2318716.2318718

[11] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 37–48. doi:10.1145/1989323.1989328

[12] Haran Boral and David J DeWitt. 1983. Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines.

[13] Amirali Boroumand, Saugata Ghose, Geraldo F. Oliveira, and Onur Mutlu. 2022. Polynesia: Enabling High-Performance and Energy-Efficient Hybrid Transactional/Analytical Databases with Hardware/Software Co-Design. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2997–3011. doi:10.1109/ICDE53745.2022.00270

[14] 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)*. 568–580. doi:10.1109/HPCA.2016.7446095

[15] Craig Chasseur and Jignesh M. Patel. 2013. Design and Evaluation of Storage Organizations for Read-Optimized Main Memory Databases. *Proc. VLDB Endow.* 6, 13 (aug 2013), 1474–1485. doi:10.14778/2536258.2536260

[16] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2012. CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory. In *Proceedings of the Design, Automation & Test in Europe Conference (DATE)*. 33–38. doi:10.1109/DATE.2012.6176428

[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 *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p27-chen-cidr20.pdf

[18] Clickhouse. 2023. https://clickhouse.com/docs/en/getting-started/example-datasets/star-schema.

[19] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query processing on smart SSDs: opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1221–1230. doi:10.1145/2463676.2465295

[20] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proc. VLDB Endow.* 13, 8 (apr 2020), 1206–1220. doi:10.14778/3389133.3389138

[21] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. 2015. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 31–46. doi:10.1145/2723372.2747642

[22] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA database - An architecture overview. *IEEE Data Eng. Bull.* 35 (03 2012), 28–33.

[23] Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy, and Jignesh M. Patel. 2022. SQLite: Past, Present, and Future. *Proc. VLDB Endow.* 15, 12 (aug 2022), 3535–3547. doi:10.14778/3554821.3554842

[24] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. 2023. Profiling Hyperscale Big Data Processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 47, 16 pages. doi:10.1145/3579371.3589082

[25] Juan Gmez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2021. Benchmarking Memory-Centric Computing Systems: Analysis of Real Processing-In-Memory Hardware. In *2021 12th International Green and Sustainable Computing Conference (IGSC)*. 1–7. doi:10.1109/IGSC54211.2021.9651614

[26] Nastaran Hajinazar, Geraldo F. Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. 2021. SIMDRAM: a framework for bit-serial SIMD processing using DRAM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 329–345. doi:10.1145/3445814.3446749

[27] Ben Hannel and Kevin Leong. [n. d.]. Rockset Performance Evaluation on the Star Schema Benchmark. https://rockset.com/Rockset_Star_Schema_Benchmark_April2022.pdf

[28] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and T. N. Vijaykumar. 2020. Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 372–385. doi:10.1109/MICRO50266.2020.00040

[29] Intel. 2023. Intel In-Memory Analytics Accelerator Architecture Specification. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[30] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an Open Framework for Query Optimization and Compilation. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2389–2401. doi:10.14778/3551793.3551801

[31] Aarati Kakaraparthy, Jignesh M. Patel, Brian P. Kroth, and Kwanghyun Park. 2022. VIP Hashing: Adapting to Skew in Popularity of Data on the Fly. *Proc. VLDB Endow.* 15, 10 (jun 2022), 1978–1990. doi:10.14778/3547305.3547306

[32] 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* 42, 1 (2022), 116–127. doi:10.1109/MM.2021.3097700

[33] Ralph Kimball and Margy Ross. 2002. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling* (2nd ed ed.). Wiley, New York.

[34] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. 2011. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *Proc. VLDB Endow.* 5, 1 (sep 2011), 61–72. doi:10.14778/2047485.2047491

[35] Daehan Kwon, Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gi-Moon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, Jungyeon Kim, Junyeol Jeon, Nahsung Kim, Yongkee Kwon, Vladimir Kornijcuk, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Guhyun Kim, Byeongju An, Jaewook Lee, Donguc Ko, Younggun Jun, Ilwoong Kim, Choungki Song, Ilkon Kim, Chanwook Park, Seho Kim, Chunseok Jeong, Euicheol Lim, Dongkyun Kim, Jieun Jang, Il Park, Junhyun Chun, and Joohwan Cho. 2023. A 1ynm 1.25V 8Gb 16Gb/s/Pin GDDR6-Based Accelerator-in-Memory Supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep Learning Application. *IEEE Journal of Solid-State Circuits* 58, 1 (2023), 291–302. doi:10.1109/JSSC.2022.3200718

[36] 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 *Proceedings of the 18th International Workshop on Data Management on New Hardware* (Philadelphia, PA, USA) *(DaMoN '22)*. Association for Computing Machinery, New York, NY, USA, Article 2, 9 pages. doi:10.1145/3533737.3535093

[37] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 43–56. doi:10.1109/ISCA52012.2021.00013

[38] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R. Stan, and Kevin Skadron. 2020. Fulcrum: A Simplified Control and Access Mechanism Toward Flexible and Practical In-Situ Accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 556–569. doi:10.1109/HPCA47549.2020.00052

[39] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. doi:10.1145/2897937.2898064

[40] 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

[41] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 289–300. doi:10.1145/2463676.2465322

[42] Yinan Li and Jignesh M. Patel. 2014. WideTable: An Accelerator for Analytical Data Processing. *Proceedings of the VLDB Endowment* 7, 10 (June 2014), 907–918. doi:10.14778/2732951.2732965

[43] 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

[44] Zezhou Liu and Stratos Idreos. 2016. Main Memory Adaptive Denormalization. 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, 2253–2254. doi:10.1145/2882903.2914835

[45] Micron Technology, Inc. 2025. NVDIMM | Micron Technology Inc. https://www.micron.com/products/memory/dram-modules/nvdimm. Accessed: 2025-06-29.

[46] Samsung Advanced Institute of Technology. [n. d.]. SAIT's PIMSimulator. https://github.com/SAITPublic/PIMSimulator.

[47] P. O'Neil, E. O'Neil, and X. Chen. 2007. The Star Schema Benchmark. http://www.cs.umb.edu/~poneil/StarSchemaB.pdf.

[48] Philippos Papaphilippou and Wayne Luk. 2018. Accelerating Database Systems Using FPGAs: A Survey. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 125–1255. doi:10.1109/FPL.2018.00030

[49] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A Data Platform Based on the Scaling-up Approach. *Proc. VLDB Endow.* 11, 6 (oct 2018), 663–676. doi:10.14778/3184470.3184471

[50] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, Amsterdam Netherlands, 1981–1984. doi:10.1145/3299869.3320212

[51] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. 2008. Constant-Time Query Processing. In *2008 IEEE 24th International Conference on Data Engineering*. 60–69. doi:10.1109/ICDE.2008.4497414

[52] Paulo C. Santos, Geraldo F. Oliveira, Diego G. Tomé, Marco A. Z. Alves, Eduardo C. de Almeida, and Luigi Carro. 2017. Operand size reconfiguration for big data processing in memory. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, David Atienza and Giorgio Di Natale (Eds.). IEEE, 710–715. doi:10.23919/DATE.2017.7927081

[53] 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

[54] 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

[55] Utku Sirin and Anastasia Ailamaki. 2020. Micro-architectural Analysis of OLAP: Limitations and Opportunities. *Proceedings of the VLDB Endowment* 13, 6 (2020), 840–853. doi:10.14778/3380750.3380755

[56] StarRocks. 2023. https://docs.starrocks.io/en-us/2.5/benchmarking/ SSB_Benchmarking.

[57] Aaron Stillmaker, Zhibin Xiao, and Bevan M. Baas. 2012. Toward More Accurate Scaling Estimates of CMOS Circuits from 180 nm to 22 nm. Univ. of California-Davis Tech. Report ECE-VCL-2011-4.

[58] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. 2014. Fine-grained partitioning for aggressive data skipping. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1115–1126. doi:10.1145/2588555.2610515

[59] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 385–394. doi:10.14778/1687627.1687671

[60] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: an intelligent storage engine with support for advanced SQL offloading. 7, 11 (July 2014), 963–974. doi:10.14778/2732967.2732972

[61] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Architecture News* 23, 1 (Mar 1995), 20–24. doi:10.1145/216585.216588

[62] Sam Likun Xi, Aurelia Augusta, Manos Athanassoulis, and Stratos Idreos. 2015. Beyond the Wall: Near-Data Processing for Databases. In *Proceedings of the 11th International Workshop on Data Management on New Hardware* (Melbourne, VIC, Australia) *(DaMoN'15)*. Association for Computing Machinery, New York, NY, USA, Article 2, 10 pages. doi:10.1145/2771937.2771945

[63] Daniel Ziener, Florian Bauer, Andreas Becher, Christopher Dennl, Klaus Meyer-Wegener, Ute Schürfeld, Jürgen Teich, Jörg-Stephan Vogt, and Helmut Weber. 2016. FPGA-Based Dynamically Reconfigurable SQL Query Processing. 9, 4, Article 25 (Aug. 2016), 24 pages. doi:10.1145/2845087

Received 14 September 2025; revised 24 November 2025; accepted 7 December 2025