



PystachIO: Efficient Distributed GPU Query Processing with PyTorch over Fast Networks & Fast Storage

Jigao Luo*
jigao.luo@tu-darmstadt.de
TU Darmstadt

Muhammad El-Hindi
muhammad.el-hindi@tum.de
Technische Universität München

Nils Boeschen*
nils.boeschen@cs.tu-darmstadt.de
TU Darmstadt & DFKI Darmstadt

Carsten Binnig
carsten.binnig@cs.tu-darmstadt.de
TU Darmstadt & hessian.AI & DFKI Darmstadt

ABSTRACT

The AI hardware boom has led modern data centers to adopt HPC-style architectures centered on distributed, GPU-centric computation. Large GPU clusters interconnected by fast RDMA networks and backed by high-bandwidth NVMe storage enable scalable computation and rapid access to storage-resident data. Tensor computation runtimes (TCRs), such as PyTorch, originally designed for AI workloads, have recently been shown to accelerate analytical workloads. However, prior work has primarily considered settings where the data fits in aggregated GPU memory. In this paper, we systematically study how TCRs can support scalable, distributed query processing for large-scale, storage-resident OLAP workloads. Although TCRs provide abstractions for network and storage I/O, naive use often underutilizes GPU and I/O bandwidth due to insufficient overlap between computation and data movement. As a core contribution, we present PystachIO, a prototype of a PyTorch-based distributed OLAP engine that combines fast network and storage I/O with key optimizations to maximize GPU, network, and storage utilization. Our evaluation shows up to 3× end-to-end speedups over existing distributed GPU-based query processing approaches.

PVLDB Reference Format:

Jigao Luo, Nils Boeschen, Muhammad El-Hindi, and Carsten Binnig. PystachIO: Efficient Distributed GPU Query Processing with PyTorch over Fast Networks & Fast Storage. PVLDB, 19(9): 2494 - 2507, 2026. doi:10.14778/3819518.3819566

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/DataManagementLab/PystachIO>.

1 INTRODUCTION

The Rise of AI Hardware. The recent AI boom has driven massive investments in new data centers built for large-scale, distributed AI training [37, 51]. These facilities increasingly adopt HPC-style architectures built around high-performance GPU clusters interconnected by fast, low-latency networks and backed by high-bandwidth NVMe storage [14, 62]. As GPUs have become the primary compute

*Equal contribution. Correspondence goes to jigao.luo@tu-darmstadt.de. This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment. Proceedings of the VLDB Endowment, Vol. 19, No. 9 ISSN 2150-8097. doi:10.14778/3819518.3819566

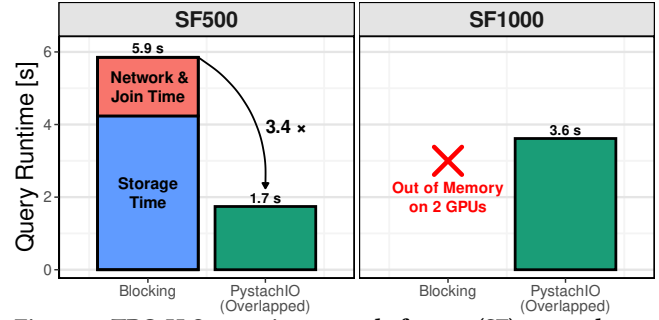


Figure 1: TPC-H Q3 runtime at scale factors (SF) 500 and 1000 on two GPUs over SSD-resident, non-co-partitioned tables. A blocking PyTorch-style baseline executes sequentially, leading to long runtimes and out-of-memory errors. In contrast, PystachIO overlaps storage I/O, networking, and computation to reduce query time by over 3×.

units for training and inference, hyperscalers increasingly rely on their high compute density and acceleration capabilities. Consequently, modern data center infrastructure enables efficient GPU-centric distributed execution and fast access to massive storage-resident datasets. Beyond AI workloads, these investments are accelerating the commoditization of large-scale GPU computing, making high-performance GPU infrastructure broadly available and opening new opportunities for other data-intensive workloads, including analytical query processing.

Leveraging AI Abstractions for Query Processing. The rapid adoption of large-scale GPU computing is also fueled by simple, expressive tensor computation runtimes (TCRs) such as PyTorch [34]. Although these abstractions were initially designed for developing and training AI models, recent work showed that they can also be used to implement analytical database systems [17]. By mapping relational operators (e.g., filters, joins, and aggregations) to the vectorized tensor operations provided by PyTorch, these systems enable high-performance processing for OLAP workloads on GPUs. **Limitations of Existing Tensor-Based Engines.** Existing TCR-based systems for analytical processing primarily target single-node setups and assume that table data fits entirely in GPU memory. However, modern OLAP workloads often exceed the memory capacity of a single GPU, making such single-node in-memory designs cost-prohibitive and impractical. Recent work has begun to explore distributed query processing with TCRs to handle larger datasets [61], but it still assumes that the workload fits into the aggregate memory of the GPU cluster. Consequently, scaling to larger workloads

requires provisioning additional accelerator nodes, which increases costs and operational complexity.

Distributed, Out-Of-Memory OLAP With TCRs. To address these limitations and enable large-scale, storage-resident OLAP on GPUs, we investigate how tensor computation runtimes can support distributed, out-of-memory query processing. In contrast to prior work, our approach leverages the full AI data center stack: fast RDMA-capable networks for distributed computation and high-bandwidth NVMe SSDs for out-of-memory execution. A key challenge is that naively reusing existing TCR abstractions for network and storage I/O, as is typical in large-scale model training, yields a blocking execution model in which data is first loaded in bulk before processing begins. While this pattern is suitable for compute-intensive, iterative training workloads, it results in low query performance and even out-of-memory (OOM) errors for I/O-intensive OLAP workloads, as shown by the left bars in Figure 1.

PystachIO: Efficient Large-Scale OLAP With TCRs. We present *PystachIO*, a prototypical GPU-accelerated distributed query engine that enables fast out-of-memory OLAP with tensor computation runtimes, as shown by the right bars in Figure 1. *PystachIO* is built on PyTorch and is designed to efficiently leverage high-performance networks and NVMe storage. Our key observation is that achieving high performance requires careful utilization of network and storage bandwidth to avoid bottlenecks and idle GPU cycles. *PystachIO* addresses this by overlapping computation with storage and network I/O, while reducing synchronization overheads and optimizing memory usage for large datasets.

Contributions and Outline. In summary, this paper makes the following contributions: (1) We analyze why naive use of TCR I/O abstractions fails to fully exploit the capabilities of fast networks and storage (Section 2). (2) We introduce optimizations that enable query processing to approach the utilization limits of network and storage bandwidth by effectively overlapping I/O with computation (Sections 3 and 4). (3) We realize these techniques in *PystachIO*, a distributed query engine that carefully coordinates storage and networking to maximize end-to-end performance (Section 5). (4) We evaluate *PystachIO* on analytical workloads, demonstrating performance close to the theoretical hardware bounds (Section 6).

We conclude by discussing related work in Section 7 and summarizing the main insights in Section 8.

2 SYSTEM OVERVIEW & CHALLENGES

In this section, we provide an overview of *PystachIO*’s system architecture and discuss the challenges of leveraging TCRs to build an efficient and scalable distributed database system.

2.1 System Architecture

He et al. [17] pioneered the idea of using AI-oriented TCRs for GPU-accelerated query processing. *PystachIO* advances this direction by implementing a distributed TCR-based engine that exploits the fast networks and storage technologies commonly used in modern AI data centers: RDMA-enabled networking and high-bandwidth NVMe storage. Figure 2 illustrates the overall system design. Like prior work, *PystachIO* employs a column-oriented storage layout in which each column is represented as a tensor, to enable the use of optimized parallel TCR primitives. The system further partitions

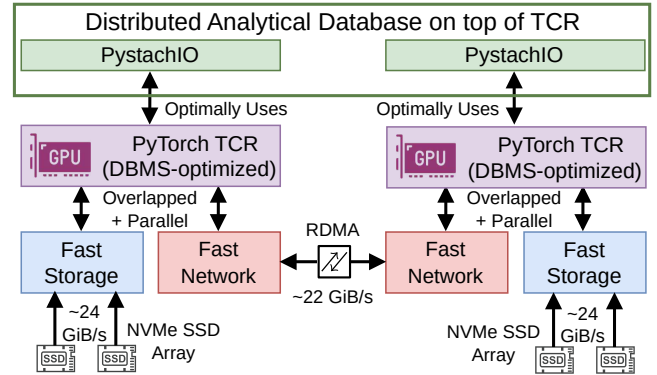


Figure 2: PystachIO: a database architecture that exploits fast network and storage hardware in AI data centers on top of the PyTorch TCR. Naive use of the TCR does not achieve optimal performance. *PystachIO* adds database-specific optimizations, including overlapping compute and I/O and parallel storage access, to enable highly efficient query processing.

these tensors horizontally into chunks, which enables overlapping I/O and computation and is key to *PystachIO*’s efficient use of fast storage and networking.

Fast Networking and Storage Using TCRs. TCRs expose abstractions for high-speed networking and storage, originally designed for distributed AI workloads. However, these abstractions do not deliver high performance for database workloads out-of-the-box, since data movement rather than computation dominates database execution. Below, we outline the relevant TCR primitives for fast networking and storage and highlight the challenges that arise when using them for database workloads. We then summarize how *PystachIO* addresses these challenges through careful use and internal extensions of TCRs. With these improvements, TCRs such as PyTorch can serve as a high-performance substrate for distributed database systems without requiring the design of systems based on low-level, system-specific interfaces [20, 21].

2.2 Fast Networking With TCRs

PyTorch provides efficient networking capabilities typically used during distributed AI training. Specifically, its distributed communications layer offers collective and point-to-point primitives, backed by the NVIDIA Collective Communications Library (NCCL) [38], which enables direct network I/O between GPU memory and RDMA-capable network hardware via GPUDirect RDMA [39]. As shown in Figure 3(a), NCCL can saturate high-speed networks and reach near-peak bandwidth when invoked through PyTorch.

Challenges in TCR-Based Query Processing. However, naively applying collectives to implement a distributed join results in sub-optimal bandwidth (see Figure 3(a) second bar) and uneven network utilization. The primary issue is that communication is blocking, leaving GPUs idle during data shuffling. While blocking communication is acceptable in AI workloads, where computation dominates the overall cost, it becomes a bottleneck for database workloads, where network transfer time is comparable to processing time.

Network I/O in PystachIO. *PystachIO* builds on PyTorch’s distributed communication layer with the NCCL backend, to shuffle data efficiently for distributed processing across GPUs over RDMA

Figure 3: TCRs provide fast networking and storage primitives: (a) Performance of PyTorch’s AllToAll collective versus a naive distributed join. (b) SSD read speed measured during the I/O stage of a scan versus the end-to-end bandwidth.

networks. Instead of relying on PyTorch’s default blocking operations, we introduce new scheduling techniques that overlap communication with computation, effectively hiding computation latency. Additionally, PystachIO optimizes runtime memory allocation and eliminates unnecessary synchronization, sustaining a consistently high network bandwidth during query execution. We discuss the design of PystachIO’s network layer in Section 3.

2.3 Fast Storage With TCRs

PyTorch-based TCRs typically access storage-resident tabular data via libraries such as pandas [11], NumPy [10], and RAPIDS cuDF [54], supporting file formats including CSV and Apache Parquet [57]. CPU-based readers like pandas and NumPy perform parsing and data movement on the CPU and thus cannot saturate NVMe bandwidth [25, 46]. In contrast, cuDF supports GPUDirect Storage (GDS) [40, 53], which enables direct I/O between NVMe SSDs and GPU memory, followed by GPU kernels for tasks such as Parquet decompression. As Figure 3(b) shows, a GDS-enabled cuDF table scan over a Parquet file can saturate a single SSD during the I/O stage. **Challenges in TCR-Based Query Processing.** However, as with networking, PyTorch- and cuDF-based applications typically perform storage I/O in a blocking fashion. As a result, kernels that process data after I/O, such as result filtering, run only after the full I/O phase has completed, leaving the SSD idle during GPU processing. This serialized execution reduces average storage bandwidth by roughly 3.5×, as shown in Figure 3(b) (right bar).

Storage I/O in PystachIO. To enable efficient storage access in PystachIO, we redesign the GPU-based table scan by replacing blocking file-at-a-time I/O with a fully overlapped pipeline that interleaves storage I/O and GPU computation. Further optimizations include metadata caching to eliminate repeated CPU-side overheads and pinned-memory-based synchronization that removes hidden GPU stalls. These techniques transform the table scan into a fully pipelined process that saturates SSD bandwidth and keeps the GPU consistently busy, enabling efficient storage utilization for TCR-based query processing. We discuss the design of PystachIO’s storage layer in Section 4.

3 DISTRIBUTED EXECUTION WITH TCRS

Traditionally, only specialized, hand-optimized DBMSs have been able to fully exploit high-bandwidth, low-latency networks, such

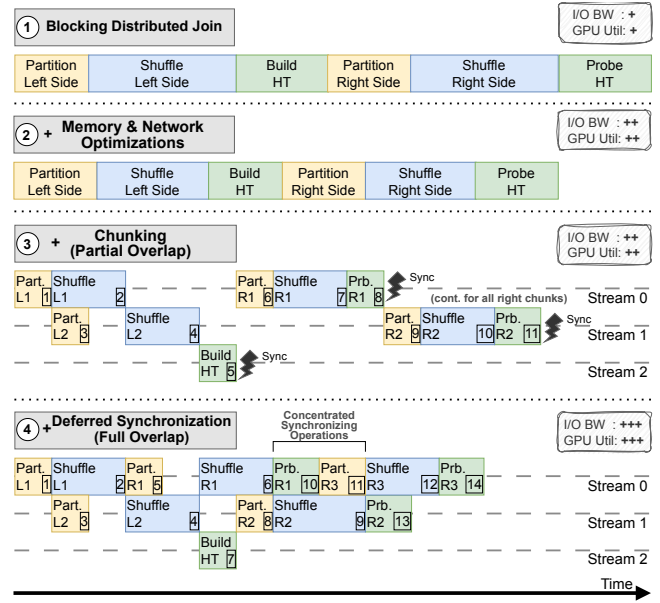


Figure 4: Distributed hash join variants in PyTorch: blocking (natural to implement in TCRs) and optimized (overlapped) approaches. Expected GPU and network I/O utilization is shown in the top-right inset for each variant. For the overlapped variants, the scheduling order of operations is annotated in the bottom-right corner.

as RDMA-capable interconnects in AI data centers (e.g., [3, 4, 12, 20, 65]). As discussed in the previous section, TCRs such as PyTorch provide high-level abstractions over networking primitives, making it easier for developers to leverage and saturate these fast networks. However, naive use of these networking abstractions in query processing underutilizes the underlying hardware. In this section, we present optimizations that enable TCR-based database systems to maximize network bandwidth by overlapping computation and communication. We then use these techniques to design highly efficient distributed hash joins, one of the most expensive operations in distributed query processing.

3.1 Naive Approach to Distributed Joins

In this section, we demonstrate how PyTorch interfaces can be leveraged to implement a distributed hash join. For the network I/O between GPUs over RDMA-capable networks, we employ NCCL. We use the *AllToAll* collective, which is implemented as a wrapper around the respective point-to-point transfers.

Anatomy of a Distributed Hash Join. A distributed hash join over two tables can be conceptually decomposed into six steps (see top of Figure 4). The local left table is first partitioned according to the join key, after which the resulting partitions are shuffled across the network. Each node then builds a hash table from the left-hand partitions it receives. The local right table undergoes the same procedure: partitioning and shuffling based on the join key. Finally, each node probes the hash table using the right-hand partitions and materializes or passes on the resulting join tuples.

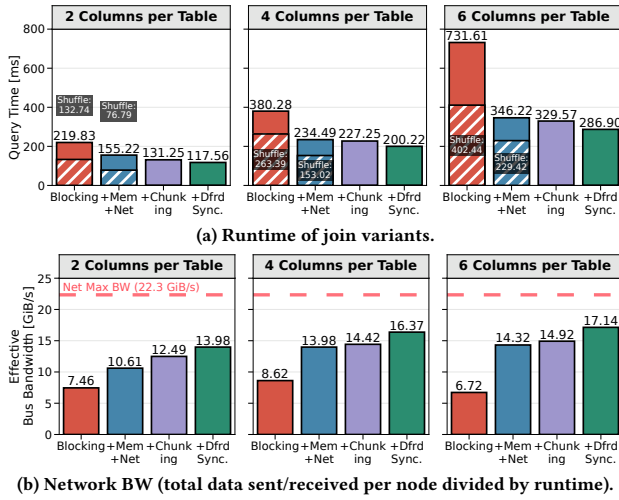


Figure 5: Distributed join performance on 2 GPU nodes for varying numbers of 8-byte columns (120M/320M build/probe tuples, hit ratio 0.5). Hatched bars show network shuffle time for the blocking variants.

A Naive Implementation in PyTorch. Using PyTorch’s tensor operations and network primitives, a straightforward blocking variant, as shown as *Blocking Distributed Join* ① in Figure 4, is simple and natural to implement. Partitioning can be performed through vectorized modulo operations combined with masked selection. To communicate partition sizes across nodes, an *AllToAll* collective can be used, preparing the receivers for incoming tensor data. The actual shuffling of columns can be implemented via NCCL’s *AllToAll* collective for each partitioned column, assuming that communication efficiently utilizes direct GPU-to-NIC I/O and fully leverages the underlying high-speed interconnect. Hash table construction and probing can be mapped to scatter and gather operations, as proposed in [17].

Analysis of the Blocking Join. We evaluate the performance of the blocking variant by shuffling tables with varying widths over a 200G RoCE network¹. We report the effective I/O-bus bandwidth, defined as the total data received (or sent) per node divided by the end-to-end join runtime. The experiment shows that the naive implementation, labeled *Blocking* (red) in Figure 5, sustains less than 50% of the maximum network bandwidth. This suboptimal performance arises from two main inefficiencies.

(1) Low Utilization of Available Network Bandwidth. Because the networking backends of TCRs are designed to balance resource usage and performance (e.g., NCCL launches only a limited number of GPU thread blocks by default), parameter tuning is required to approach the bandwidth limits of high-speed networks. In our NCCL-based setup, we tune both the per-message size (512 KiB) and the number of GPU-executed blocks (32 NCCL channels). This tuning enables us to achieve near-peak network utilization, as observed in the *AllToAll* results in Figure 3.

(2) Runtime Memory Allocation in PyTorch. PyTorch allocates memory dynamically by default. While this behavior is well-suited

for compute-bound AI training, database operators are often less compute-intensive and therefore more sensitive to memory allocation overhead. Thus, we enable the allocation of dynamically sized, data-dependent tensors by configuring PyTorch to use the RAPIDS Memory Manager (RMM) allocator [55]. RMM allocates a large block of device memory once and then serves smaller allocations from that pool with low overhead.

Performance Impact of Low-Level Optimizations. The low-level optimizations, tuning network parameters, and improving memory allocation can be applied without modifying the join algorithm itself, as shown in Figure 4 ②. They already reduce the runtime of the naive blocking join by 1.5–2×, as indicated by *+Mem+Net* (blue) in Figure 5. Because they require no algorithmic changes, they are an effective first step in optimizing distributed joins.

3.2 Optimization: Chunking

As shown earlier, a naive blocking approach for distributed joins underutilizes the bandwidth of modern networks. Hence, we investigate how to implement a non-blocking distributed join in PyTorch. Instead of blocking, we overlap communication and computation to improve network utilization and reduce overall query runtime.

Partial Overlap Using Chunking. The compute and memory resources of modern data center GPUs enable concurrent execution of network communication and local computation. We leverage this capability to design an overlapped variant of the distributed join that improves end-to-end runtime. To overlap I/O and computation, all phases of the distributed join must operate on data chunks rather than full-column tensors. Chunking enables, for example, the partitioning of one chunk to run concurrently with the network shuffling of another. When implemented effectively, such an overlap significantly reduces idle periods and shortens overall processing time, as we show later.

Scheduling Overlap With GPU Streams. To overlap different phases of the join, we leverage the ability of modern GPUs to schedule operations across multiple (CUDA) streams. These streams enable the asynchronous launch of compute kernels and network primitives, allowing operations issued to different streams to run concurrently, purely from GPU internal overlap, without any CPU thread parallelism. As illustrated by *Chunking* ③ in Figure 4, we use standard data-parallel execution across streams: for each data chunk, partitioning, shuffling, and hash table operations are scheduled sequentially within one stream, while other chunks are assigned to separate streams. For example, Figure 4 shows how operations on two chunks of the left-side table can overlap: the shuffling of the first left chunk L1 and the partitioning of L2 are executed concurrently. Because the hash table is built on a fully materialized table [17], we build it once per node in a separate stream after all left-side chunks have been received. The scheduling order for this variant (annotated as numbers in the lower right corner) follows the per-chunk pipeline: for chunk R1, partition, shuffle, and probe are scheduled back-to-back. The performance impact of overlap via *Chunking* over the optimized blocking approach is modest, as shown by the purple bars in Figure 5.

However, since this approach enables *streaming* received chunks to subsequent operators and dropping data after processing (keeping only the very necessary intermediate results in accelerator

¹All details on the hardware setup are provided in Section 6.

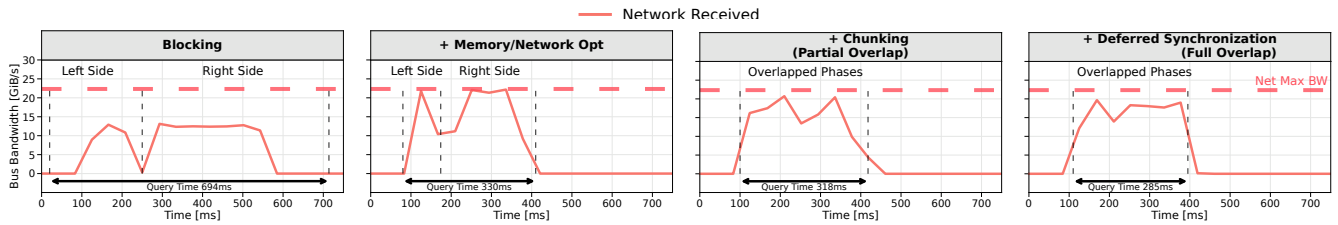


Figure 6: Timeline of bandwidth monitoring of one involved NIC during different distributed join variants. 2 Nodes, 120M/160M build/probe side tuples in total, 6 Columns, join hit ratio 0.5.

memory), chunking allows for relieving memory pressure drastically, and consequently to join large tables in fewer accelerators.

Remaining Bandwidth Limitations. The achieved bandwidth still falls short of the network’s capacity. This observation is evident from the network utilization timelines in Figure 6, which show the activity of a single NIC during the shuffling process. As expected, the *Blocking* variants (first two timelines) exhibit pronounced gaps between processing the two input tables, during which network traffic temporarily drops. The *Chunking* variant eliminates this large gap and reduces overall runtime, yet still shows considerable bandwidth fluctuations, especially in the second half of execution, indicating inefficient network utilization.

Stream-Synchronizing Operations Limit Overlap. One root cause of the remaining bandwidth underutilization is reduced overlap between operations due to implicit stream synchronization. Synchronizing a GPU stream blocks the CPU until all previously enqueued operations (kernels or memory transfers) in that stream have completed. Many TCR primitives introduce such synchronizations implicitly because they must return tensors with data-dependent metadata (e.g., size) that is only known after execution on the device. A representative example is `masked_select`, which selects tensor entries based on a boolean mask and is used for partitioning as well as for hash table probing [17]. Its output size depends on the mask, but runtimes like PyTorch require a correctly sized tensor at return, even if data is filled asynchronously. The TCR therefore synchronizes the stream and materializes metadata on the CPU to determine size, blocking further kernel launches from the CPU thread. Such synchronizations are inherent to distributed query processing, which involves operations with data-dependent result sizes (e.g., selections, joins, group-by operations). In the case of the distributed join, the partitioning and hash table operations include stream-synchronizing primitives, while the NCCL *AllToAll* collective itself can execute asynchronously (after metadata has been exchanged).

These synchronizations are particularly harmful during hash table probing on received chunks. If probing is scheduled per chunk directly after shuffling, the synchronizing probe step blocks until all earlier operations in that stream (partitioning and shuffling) have completed, as illustrated by the lightning symbol for the probe of R1 in Figure 4 (*Chunking* ③). This blocking limits inter-stream overlap by preventing the CPU from enqueueing additional work to other streams. In the timeline for this variant (Figure 6, third plot), this behavior appears as bursts of NIC activity separated by idle periods, reflecting the reduced effectiveness of overlap.

3.3 Optimization: Deferred Synchronization

In TCR-based processing, stream-synchronizing operations are unavoidable for data-dependent operations across streams, making achieving overlap a unique challenge in this setting. However, we can reduce the synchronization overhead and achieve better overlap by accounting for relevant operations during scheduling. Our key idea is to front-load operations with implicit stream synchronization at the beginning of the work scheduled per stream. For example, instead of scheduling partitioning, shuffling, and probing for the right-hand chunk R1 back-to-back in the same stream, we defer probing R1 to avoid forcing an early stream synchronization (see the lower part of Figure 4). This strategy ensures that when another stream is used to process the next chunk R2, the asynchronous operations of the previous chunk (e.g., the network shuffle) run concurrently with the synchronizing operations of the current chunk. In the example, the probe on R1 is scheduled as the 10th operation in stream 1, before partitioning and shuffling R3. We refer to this strategy as the *Deferred Synchronization* variant ④. In this variant, hash table operations are deferred: instead of executing them immediately after shuffling within the same stream of the current chunk, they are scheduled as the first operations of the *next* chunk in that stream (see the scheduling order in the lower-right corner of Figure 4). This is most beneficial for probing each received chunk of the right-hand table, as described above, but similar measures allow the hash table build to run concurrently with operations on right-hand chunks (e.g., scheduled as the 7th operation after starting the first right-hand shuffle in the figure).

Effects of Deferred Synchronization on Overlap. Deferred synchronization enables consistent overlap between computation and communication, as visible in the rightmost timeline in Figure 6. This leads to a substantial runtime reduction and higher effective bus bandwidth, as shown for *Deferred Synchronization* (green bar) in Figure 5. Compared to the blocking variant, deferred synchronization improves performance by up to 2.5×. For larger numbers of columns (i.e., greater shuffle volume), the *Deferred Synchronization* variant approaches the practical upper limit of the network bandwidth, achieving more than 17 GiB/s out of a possible 22.3 GiB/s.

Remaining Bandwidth Limitations. To analyze the remaining performance gap to the theoretical maximum bus bandwidth of the 200 Gbit/s link, we evaluate the overlapped join variants across different transfer sizes. As shown in Figure 7, the NCCL *AllToAll* collective alone can reach close to the maximum bus bandwidth for transfer sizes of about 64 MiB (blue line). However, once the execution stream includes data partitioning, the effective bus bandwidth drops to around 18 GiB/s (orange line) and plateaus for larger

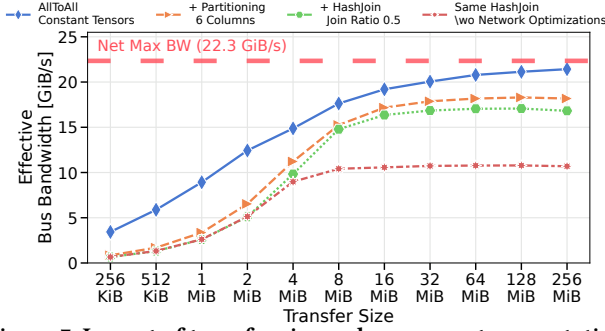


Figure 7: Impact of transfer size and concurrent computation for a fully overlapped distributed join for 2 Nodes: Bus bandwidth starting from constant-sized transfer without computation, adding concurrent partitioning and join operations. The optimized join (green) is close to the pure shuffle with All-to-All (blue line) = upper baseline.

transfer sizes. Running other join phases (build and probe) concurrently with the NCCL kernel further reduces the observed bandwidth slightly (green line). This reduction stems from the NCCL collective kernel slowing down when it shares GPU resources with other compute kernels (in our case, partitioning), a known form of resource contention when NCCL and computation kernels run concurrently [48]. We partially mitigate this effect by increasing the number of thread blocks (NCCL channels) launched as part of our network-parameter optimization, as evidenced by the higher orange and green lines compared to the same join without this change (lower red line). However, this analysis reveals that the reduced bandwidth is inherent to the concurrent execution of network I/O and computation and cannot be fully mitigated.

3.4 Benefits of Chunk-Based Execution

Beyond enabling overlap, our chunk-based execution strategy offers additional benefits over a naive blocking approach.

Reduced GPU Memory Requirements. A key advantage of chunk-based execution is its reduced GPU memory footprint during distributed joins. As long as intermediate results and per-chunk buffers fit into accelerator memory, a chunk-based join can support very large table sizes. As shown in the left plot of Figure 8, for increasing probe-side table sizes, our chunk-based variants can handle substantially larger probe tables than the blocking baseline due to their pipelined execution model. Moreover, the overlapped approaches exhibit nearly linear query-time scaling as probe-side data grows, while maintaining a consistently high effective bus bandwidth of approximately 17.2 GiB/s. For reference, the gray dashed line indicates the theoretical lower bound on runtime, calculated as the per-node shuffle volume divided by the maximum network bandwidth of 22.3 GiB/s. The remaining gap between the fully overlapped approach and this lower bound is due to the slowdown of collective kernels, as discussed earlier.

Scalability With Increasing Node Counts. We also examine scalability with respect to the number of nodes in the distributed setup, keeping the probe- and build-side table sizes fixed. The right side of Figure 8 shows that the overlapped join variants scale effectively as more nodes are added because of their efficient use

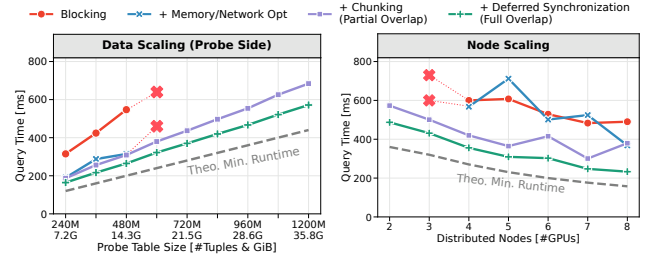


Figure 8: Scaling of a distributed join with four columns and hit ratio 0.5. Left: 2 nodes, 120M build tuples, increasing probe size. Chunking supports arbitrarily large probe tables as long as intermediates fit in memory. Right: 240M/840M build/probe tuples and increasing node count. Blocking variants are memory-constrained and slower than the fully overlapped join across all node counts.

of collective communication primitives. The performance of the *Deferred Synchronization* variant approaches the theoretical lower bound (gray dashed line) closely. Similar to the memory-capacity effect discussed earlier, the blocking join fails to execute at lower node counts because the per-node data volume exceeds the available GPU memory. For larger node counts, the blocking variants exhibit high runtime variance (e.g., blocking with memory/network optimization, blue line), as slow computation on some nodes leads to stragglers, whereas the fully overlapped variant shows more stable performance.

4 EFFICIENT STORAGE ACCESS WITH TCRS

Modern NVMe devices offer high I/O performance at a low cost and are thus also used as primary storage for AI workloads. However, efficiently exploiting NVMe in TCRs remains under-explored, as TCR research has primarily focused on compute-intensive AI workloads. In this section, we explore how to effectively integrate NVMe storage into TCRs to implement a table scan operator that reads table files from NVMe and filters tuples using predicates via filter pushdown. However, as storage interfaces in TCR are blocking, a table scan first needs to load the entire input into GPU memory before any subsequent operation, such as a join, can be applied, resulting in overall non-optimal execution. Within the following, we study how an efficient table scan can be implemented on top of PyTorch. Beyond overlapping I/O and computation, we also study how compression and pruning can further accelerate scanning.

4.1 Naive Approach to Table Scans

We now discuss how a first version of a table scan can be realized using PyTorch. In PystachIO, we store tables in columnar Parquet as a popular open file format. However, the findings and design decisions we discuss in this section enable efficient table scans with TCRs in general (for other file formats as well).

A Naive Table Scan. A first (naive) approach to implement a table scan is to read table data using an existing Parquet reader. In our case, we use the GPU-based Parquet reader from RAPIDS cuDF [54], which supports GPUDirect Storage (GDS) [40, 53] for efficient direct I/O and integrates with the PyTorch TCR. Conceptually, a CPU thread issues the `read()` in three sequential phases, illustrated as ① *Blocking Parquet Reader* in Figure 10. In the first phase, a CPU thread

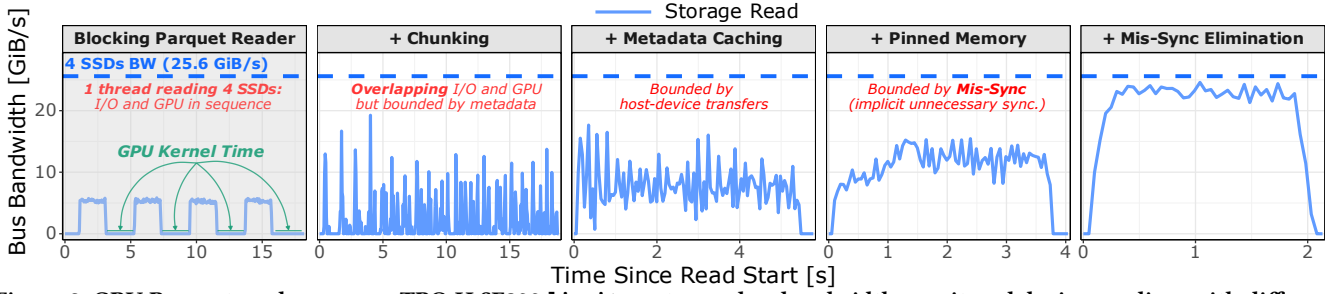


Figure 9: GPU Parquet reader scan on TPC-H SF300 lineitem: storage bus bandwidth monitored during reading, with different optimizations. 8 threads are used in all cases except for the blocking reader.

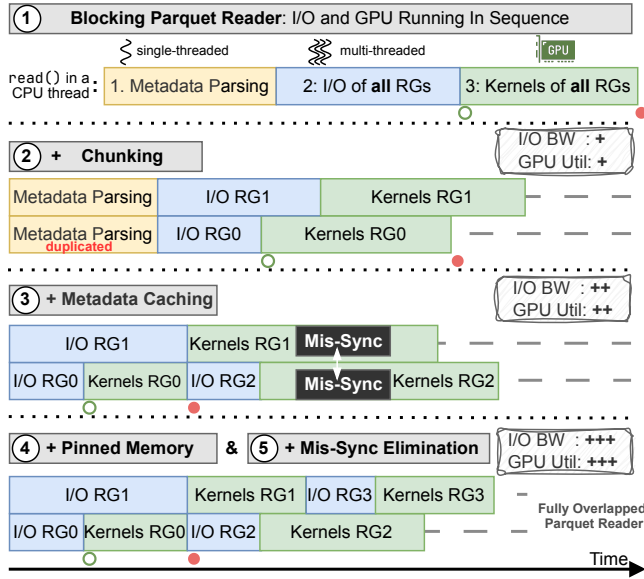


Figure 10: GPU Parquet reader: from blocking to fully overlapping storage I/O and GPU with different optimizations

parses the Parquet file’s metadata [1, 16, 26]. Second, multiple CPU threads from a pool are used to transfer the requested RowGroups (RGs) to GPU memory for saturating storage bandwidth [5, 36]. Finally, GPU kernels decompress and decode the RGs loaded in GPU memory, producing an uncompressed columnar representation for downstream query processing. The kernels are launched into the CUDA streams assigned to the CPU thread issuing the `read()` call. **Analyzing the Naive Table Scan.** We identify several limitations of the blocking design used by the current GPU Parquet reader. First, I/O and GPU execution are strictly serialized: the GPU remains idle during I/O and can launch kernels only afterward, as indicated by $\circ$ in Figure 10. As a result, consumer operators (e.g., a join) can begin only after entire columns have been fully materialized in GPU memory, marked with $\bullet$ in Figure 10. Second, the reader first loads all requested RGs into GPU memory before decompressing and decoding them, placing substantial pressure on the limited GPU memory and increasing the risk of OOM errors. Third, the reader can only read one file at a time, limiting the ability to utilize multiple SSDs. In Figure 9, we monitor the storage bus while scanning four files on four separate SSDs using *Blocking Parquet Reader* with one thread. Each SSD achieves a peak sequential read bandwidth of 6.4

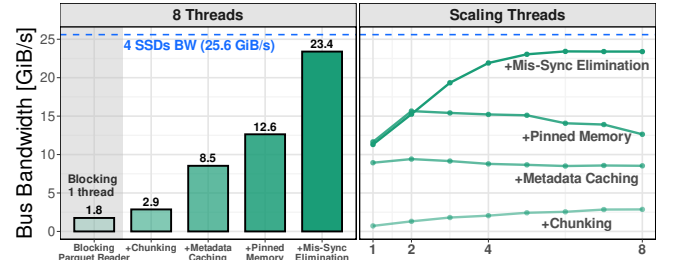


Figure 11: GPU Parquet reader scan on TPC-H SF300 lineitem: average storage bus bandwidth under different optimizations. Left: 8 threads are used in all cases except for the blocking reader. Right: Thread scalability from 1 to 8.

GiB/s. Due to the blocking design and the restriction to reading one file at a time, only one SSD is active at any given time during our monitoring. Furthermore, during the GPU kernel phase, the storage bus is completely idle in turn. Overall, when using the *Blocking Parquet Reader* to implement a table scan, sequential file reading followed by GPU kernel execution for decompression and decoding, the achieved average storage bandwidth is only 1.8 GiB/s, as shown on the left grey side of Figure 11.

4.2 Overlapping Computation and Storage I/O

To address the limitations of the naive (blocking) table scan for Parquet files, we propose a new Parquet reader for TCRs with all optimizations illustrated in Figure 10, to fully overlap storage I/O and GPU computation. In the following, we first demonstrate that, similar to networking, overlapping is crucial for reducing existing overhead in current GPU query engines with storage capabilities [23, 24] and afterward discuss further important optimizations. **Chunking.** Chunking enables reading column data at finer granularity rather than loading entire columns at once [5, 36], forming the basis for overlapped I/O and enabling all subsequent optimizations discussed later. As shown in ② of Figure 10, *Chunking* enables multiple CPU threads to issue `read()` on different chunks, with each thread owning independent CUDA streams for launching kernels. In addition, chunking also allows CPU threads to issue parallel file reads across multiple SSDs. Another important benefit is reduced memory pressure, since chunks are pipelined directly into consumer operators [35] rather than materializing an entire column in GPU memory. An interesting side effect is that chunking relaxes the row-limit constraints of cuDF, which otherwise restrict the maximum supported read size [59, 60]. As illustrated in Figure 9,

Chunking, unlike blocking, enables sustained I/O bandwidth by overlapping computation with data transfer and by reading in parallel from multiple SSDs. Although *Chunking* allows short bursts of I/O bandwidth reaching up to 20 GiB/s in Figure 9, chunking alone does not maintain consistently high throughput: the average bandwidth reaches only 2.9 GiB/s, as shown in the *Chunking* configuration in Figure 11. Next, we discuss additional optimizations that enable constant high-throughput I/O in the table scan.

Metadata Caching. Metadata in tabular file formats describes the table schema and provides information that enables data pruning, i.e., identifying prunable RGs from the file. An inherent overhead of the naive reader is that metadata is re-read in every read request. For AI workloads, this overhead is negligible because the read sizes are typically large and not chunked, and the workloads are compute-intensive. However, for database workloads, particularly under chunked reads, this results in repeatedly reading and parsing the same metadata for each chunk. As illustrated in ② *Chunking* in Figure 10, the same table metadata is redundantly read for every chunk of the same table. In PystachIO, we address this inefficiency by caching metadata upfront, thereby decoupling it from the read path. Our approach also aligns with common practice in CPU-based Parquet readers [9, 15]. Performance-wise, as shown in Figure 9, *Metadata Caching* significantly increases sustained I/O bandwidth by decoupling metadata from the critical path. With caching enabled, the average read bandwidth reaches 8.5 GiB/s, as shown under *Metadata Caching* on the left of Figure 11.

Pinned Memory Usage & Mis-Sync Elimination. After caching metadata, a new performance bottleneck surfaced: unstable multi-stream performance caused by unnecessary synchronizations between CUDA streams. These superfluous synchronizations disrupt parallel execution and leave the GPU idle for short periods, as illustrated by the black box in ③ of Figure 10. This overhead is deeply embedded in the reader, and similar issues have been reported in libcudf-based systems such as RAPIDS accelerated Spark and cuDF-Velox [23, 24].

Our analysis shows that these unnecessary synchronizations (mis-sync) are triggered by transfers between pageable CPU memory and GPU memory, for example, when copying small control metadata or element counts typically below a few hundred bytes. Although each transfer is tiny, a single such operation in one stream may propagate stalls to other streams [43, 44], which is particularly detrimental under our highly parallel chunked execution model. Because these transfers occur along the control path and involve only small data movements, they are usually overlooked; consequently, no prior system has pushed multi-stream reading as aggressively as we do, and this bottleneck has remained unaddressed. We further identified Thrust [45] as the primary source of mis-sync, producing millions of such synchronization triggers during multi-stream reading. To eliminate this bottleneck and achieve full GPU saturation, we first enabled CUDA pinned memory and then replaced all mis-sync-inducing operations with equivalent implementations using the low-level CUB library [42], thereby avoiding all pageable CPU memory. We follow this order, as pinned memory is a prerequisite for eliminating mis-sync. These optimizations yield a streamlined read path with fully overlapped storage I/O and GPU computation, as shown in Figure 10 (④ *Pinned Memory* and ⑤ *Mis-Sync Elimination*), resulting in the *Fully Overlapped Parquet Reader*.

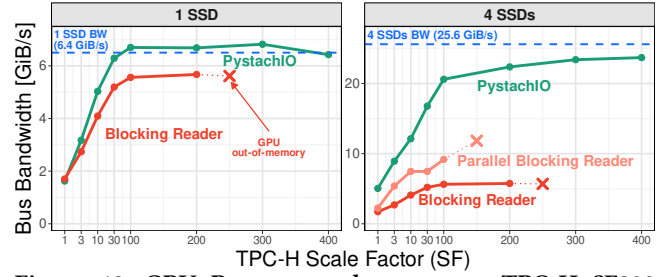


Figure 12: GPU Parquet reader scan on TPC-H SF300
lineitem: scalability across multiple datasets, compared with the blocking reader and a 4-thread parallel blocking reader.

Monitoring the PCIe bandwidth of *Pinned Memory* in Figure 9 shows that simply enabling pinned memory stabilizes I/O throughput at around 10 GiB/s, both significantly higher and more stable than before. However, as illustrated on the right side of Figure 11, the scalability of *Pinned Memory* alone holds only from one to two threads; beyond that, adding threads reduces average read bandwidth. This degradation directly reflects the mis-sync issue: with more threads, mis-sync becomes more frequent, severely hurting throughput and even leading to negative scaling. After applying *Mis-Sync Elimination*, storage bandwidth scales nearly to the full aggregated bandwidth of four SSDs, as shown in Figure 9 (*Mis-Sync Elimination*). Correspondingly, thread scalability in Figure 11 improves dramatically and becomes near-ideal.

4.3 Efficient Storage Engine With TCRs

With all optimizations enabled, our fully overlapped Parquet reader serves as the core of PystachIO’s storage engine. We believe these contributions advance the design of GPU- and TCR-based storage engines. Several optimizations have already been partially merged into libcudf [28, 29, 31] and have also motivated broader adoption [58] among CUDA library developers.

Scalability with Dataset Size. To demonstrate PystachIO as a TCR-backed storage engine, we have evaluated how PystachIO’s storage engine scales with increasing TPC-H dataset size. As shown in Figure 12, PystachIO consistently saturates the available storage bandwidth in both the 1 SSD and 4 SSDs configurations. At very small scale factors, the input is too small to fully utilize the available bandwidth. In contrast, the blocking reader processes files sequentially, one at a time, and thus achieves substantially lower bandwidth, with limited ability to benefit from multiple SSDs. Moreover, parallelizing the blocking reader yields only limited further gains. Four parallel threads performing blocking reads across four SSDs can also reach 10 GB/s at SF100, but the parallel blocking reader runs out of memory earlier than the standard blocking reader. This is expected, since in the blocking regime without chunking, higher read throughput also requires substantially larger memory buffers for decompression and decoding.

5 END-TO-END DESIGN WITH TCRS

Having demonstrated that TCR-based query processing can efficiently utilize fast networks and storage in isolation, we now investigate how to integrate these insights into an end-to-end TCR query engine that can run distributed queries on SSD-resident data.

5.1 Naive Integration of Network & Storage I/O

A naive integration of both storage and network would be to run Parquet scans from SSDs and distributed processing over RDMA as described in Section 4 and 3 in sequence, starting networking once a full table is materialized in GPU memory. To illustrate the impact of this suboptimal approach and further optimizations, we measure the achieved I/O bus bandwidth of different strategies while executing TPC-H query 3 (Q3) as a running example (since it is both storage- and networking-intensive).

Blocking Strategy. As a baseline, we execute the query using a single CPU thread that sequentially performs storage and network I/O, as illustrated in Figure 13, top row ①. The bandwidth measurement in the top-left plot of Figure 14 shows that this naive strategy does not saturate the bandwidth of fast I/O devices due to a suboptimal interface usage and missing compute and communication overlap, as shown in previous sections. This blocking strategy yields a query runtime of 6 seconds.

Applying Storage and Network I/O Optimizations. First, we apply the network and storage optimizations introduced in Sections 3 and 4 to the blocking strategy, as visualized by ② in Figure 13. The top-middle plot in Figure 14 shows the resulting performance. While the storage bandwidth reaches the device limit, and the query runtime is reduced to 3.3s, we can observe two remaining challenges: (1) The storage bandwidth shows oscillations with repeated up-and-down patterns. (2) The network bandwidth still does not reach the device limit. To address these challenges, we introduce two further optimizations that are integral for end-to-end engine performance.

5.2 Storage Optimization: Reader Combining

The previously observed oscillations in storage bandwidth occur because each table scan operator is executed by a separate Parquet reader with its own storage threads, as described in Section 4. This design serializes table scans: the start and completion of each scan cause corresponding rises and drops in bandwidth. To address this issue, *reader combining*, illustrated as ③ in Figure 13, coordinates table reads via a shared storage I/O thread pool. Reader combining issues read requests for all table scans in a query plan concurrently, eliminating bandwidth fluctuations and sustaining near-maximum storage bandwidth, as indicated in blue in the top-right plot of Figure 14. A key insight is that global awareness of the entire query plan, rather than optimizing single operators in isolation (e.g., the table scan), is crucial for efficient TCR-based query processing.

5.3 Network Optimization: Dynamic Buffering

We now address the second inefficiency, the underutilization of network bandwidth. It arises from transferring data chunks with variable and unpredictable sizes, due to the distributed joins consuming the output of preceding operators (e.g., filters), whose output size depends on their selectivity. As shown in Figure 7, network utilization drops once chunk sizes fall below a few MiB. To avoid small chunks and stabilize the transferred chunk size, we introduce a dynamic buffering operator (see ④ in Figure 13), denoted as *dynamic buffering*. We place it before chunk-size-sensitive operators, such as the distributed join, to aggregate incoming chunks until a sufficiently large buffer size is reached. Figure 14 shows the performance impact of dynamic buffering: the network bandwidth

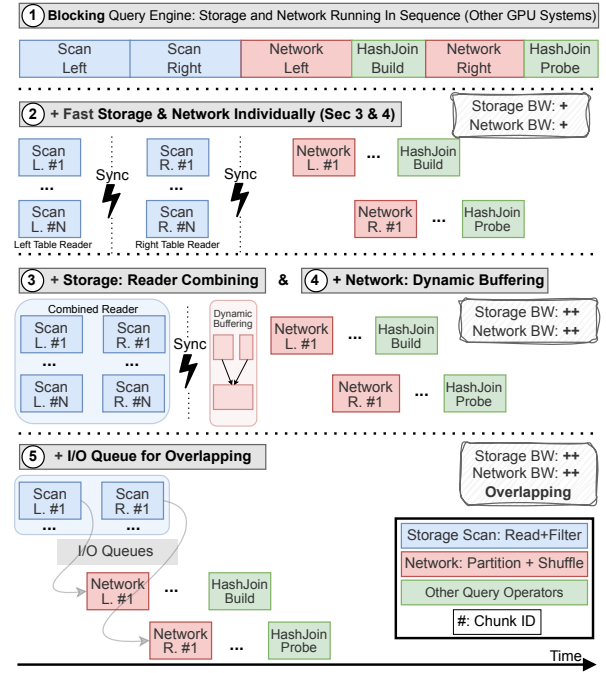


Figure 13: Query processing in TCRs: from fully blocking to fully overlapping storage, network I/O, and query execution with different optimizations.

increases from 11.5 GiB/s (top-middle) to 15.5 GiB/s (top-right, in red), thereby saturating the overlapped shuffle bandwidth limit established in Section 3.

5.4 Overlapping Network and Storage I/O

The previous sections have shown that integrating network and storage I/O for end-to-end TCR-based query processing requires more than simply executing them sequentially. Additional optimizations are needed to fully utilize the bandwidth of both subsystems (Figure 14, top). PystachIO aims to avoid sequential computation and I/O by overlapping both. This allows for reducing memory pressure by discarding fully-processed chunks, keeping only necessary intermediate results.

Main Challenges of Overlap. We identify two main challenges when overlapping network and storage I/O with TCRs. First, TCR distributed collective operations require execution in the same order across all nodes. If collective primitives are initiated from different threads, this would require additional synchronization, degrading performance and undermining the simplicity of TCRs. Therefore, approaches such as a global I/O thread pool shared by network and storage are not feasible. Second, network and storage I/O have different control-path characteristics. While network shuffling can be driven efficiently by a single thread, storage I/O for a Parquet table scan typically requires multiple threads to saturate SSD bandwidth (see Sections 3 and 4). Because the table scan typically serves as the base operator in query execution, careful coordination is necessary to ensure sufficient resources (e.g., memory) are available for downstream tasks, such as data shuffling. PystachIO employs two techniques to address these challenges: *queue-based scheduling* and *adaptive regulation*.

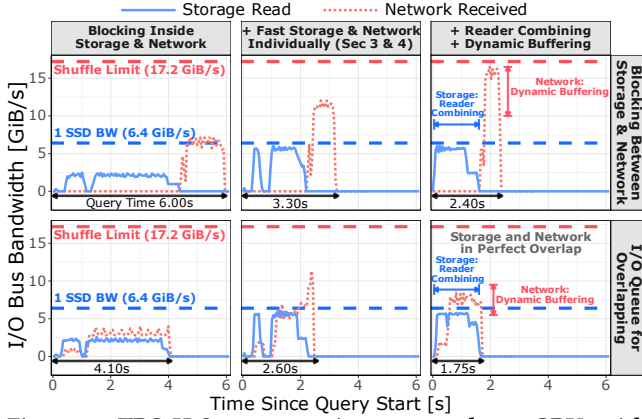


Figure 14: TPC-H Query 3 runtime executed on 2 GPUs with I/O performance: storage and network I/O bus bandwidth monitored over the query time with different optimizations.

Queue-Based Scheduling. To address the first challenge and avoid unnecessary thread synchronization, we keep the network and storage I/O paths separate and connect them via a shared multi-threaded queue, as illustrated by ⑤ in Figure 13. Such a queue is not provided as a TCR primitive because standard TCR blocking I/O is executed in a single thread and requires no coordination. PystachIO therefore implements a new multi-producer, single-consumer queue as a PyTorch component, which allows multiple CPU threads in the Parquet reader to enqueue filtered data chunks into GPU memory, while a dedicated networking thread for data shuffling dequeues the chunks for distributed query processing. With this design, network and storage I/O can progress independently, allowing for full overlap between the two.

Performance Impact of Overlap. To quantify the impact of overlap on query execution performance, we analyze I/O-bus bandwidth in the prior experiments with overlap enabled, as shown in the bottom row of Figure 14. Enabling overlap with unoptimized I/O (Figure 14, bottom-left) reduces the query runtime from 6 s to 4 s, even though I/O remains underutilized. Adding the individual I/O optimizations (Figure 14, bottom-middle) increases storage bandwidth utilization but still suffers from the issues that motivate reader combining and dynamic buffering techniques. Combining queue-based scheduling with these optimizations (Figure 14, bottom-right) reduces the query runtime to 1.75 s. In this variant, network bandwidth is limited by the preceding storage I/O and therefore stabilizes at 8 GiB/s, which is sufficient to almost completely hide the networking phase. For the queue variants, data received from storage and network shares the same PCIe connection to the GPUs. However, this potential bottleneck is mitigated since data is loaded in compressed form, and because filter pushdown ensures that shuffled data is typically smaller than data ingested from storage.

Adaptive Regulation. Beyond decoupling control paths and overlapping I/O, our queue-based scheduling also enables *adaptive regulation* to prevent the second challenge of resource imbalances and GPU out-of-memory (OOM) failures. OOM can occur in GPU-based query processing when I/O loads data into GPU memory faster than compute operators can consume it. To mitigate the issue, adaptive regulation associates each I/O queue with a table and assigns a maximum queue size during query planning. When a queue reaches its

limit, storage threads block before producing additional chunks (as a form of backpressure), throttling storage reads if necessary and indirectly aligning the speeds of storage, network, and compute. PystachIO can robustly process datasets larger than GPU memory without triggering OOM errors in TCRs with adaptive regulation, incurring negligible storage bandwidth costs, as we will show in the evaluations in the following section.

6 END-TO-END EVALUATION

To quantify the benefits of PystachIO’s optimizations end-to-end, we test its performance and scalability with the TPC-H benchmark.

6.1 Experimental Setup

We first summarize common experimental settings.

Hardware. All experiments are run on an NVIDIA DGX system [8] with Ubuntu 22.04 (DGX OS 6), Linux kernel 5.15, CUDA 12.9, and PyTorch 2.10. The system has two AMD EPYC 7742 64-core CPUs, 8 NVIDIA A100 GPUs with 40 GiB memory each, 8 Mellanox ConnectX-6 200G InfiniBand/Ethernet NICs, and 8 NVMe SSDs with up to 6.4 GiB/s read bandwidth. Each GPU is paired with its nearest NIC and SSD for network and storage I/O. Unless noted otherwise, we use 8 CPU threads per SSD for Parquet scans with filter pushdown. To emulate a distributed environment, we force all inter-GPU transfers to use the NICs and the 200G network.

Workload. We use TPC-H to evaluate end-to-end distributed query execution over SSD-resident data. We implement all TPC-H queries except Q20, Q21, and Q22, which contain query patterns not yet supported by PystachIO. Unless noted otherwise, each query uses a distributed symmetric repartitioning hash join for the join between its two largest tables. These tables are stored as horizontally partitioned Parquet files across the SSDs of different GPUs, while the remaining non-shuffled tables are replicated. All data is read from SSDs via GDS using Parquet scans with cached metadata and a file configuration optimized for GPU reading [30]. For queries with string attributes, we dictionary-encode strings as integers and adapt the queries accordingly.

Baselines. We compare PystachIO’s optimized overlapping approach against three blocking variants with increasing levels of optimization: (1) *Fully Blocking*: Single-threaded execution for both storage and networking. (2) *Fast Network & Storage*: All network and storage optimizations from Sections 3 and 4. (3) *Reader Combining & Dynamic Buffering*: Additionally enables reader combining and dynamic buffering from Section 5.

As a reference, we estimate that the runtime of a system combining the distributed TCR query execution described by Wu et al. (column-granular blocking execution) with a previous highly efficient storage phase would result in a performance close to baselines (2) and (3). In the final experiment, we compare PystachIO against the performance of the systems DuckDB, ClickHouse, and Sirius.

Performance Upper Bound. As an upper bound on performance, we compare PystachIO against a *theoretical minimum query runtime* T_{min} , derived from the data size and hardware bandwidths:

$$T_{min} = \max \left(\frac{\text{SSDReadSize}_{agg}}{\text{SSDReadBW}_{agg}}, \frac{\text{NetRecvSize}_{node}}{\text{NetBW}} \right) \quad (1)$$

The first term of Equation (1) bounds the runtime by the time to read the total compressed input SSDReadSize_{agg} from storage at

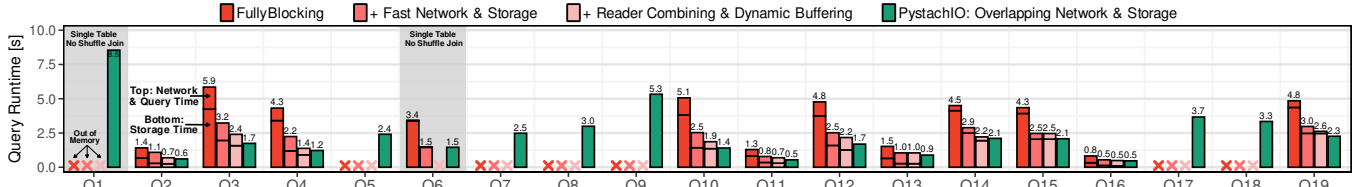


Figure 15: TPC-H SF500 runtime breakdown with 2 GPUs. Blocking variants are shown in three shades of red, with each bar decomposed into network and query-processing time (top portion) and storage table scan time (bottom portion).

full aggregated SSD bandwidth $SSDReadBW_{agg}$. For example, with 12.8 GiB of compressed data and two SSDs providing a total read bandwidth of 12.8 GiB/s (6.4 GiB/s each), the minimum runtime is 1 second. As we scale to more nodes, query runtime can become network-bound and dependent on how much data $NetRecvSize_{node}$ a node can receive at network speed $NetBW$. This bound corresponds to a system that processes queries at data-loading speed, with no decompression, network, or query-processing overhead, and does not consider contention of the GPUs’ PCIe links, which makes this a strong upper bound of performance.

6.2 Query Runtime Breakdown

We first evaluate PystachIO’s performance on all supported TPC-H queries to provide an overview using 2 GPUs on SF500 (corresponding to a 500 GB dataset). Figure 15 compares PystachIO’s approach to fully overlapped storage and network I/O against the runtime breakdown of the three blocking variants. We observe that: (1) Optimizing storage and network I/O in isolation already reduces query runtimes substantially (configuration 2). (2) Storage and network & query time often account for comparable fractions of the total runtime, indicating that overlapping these phases offers substantial gains. (3) PystachIO’s query runtime is only slightly higher than the storage time of the fastest blocking variant (configuration 3) and is close to the maximum of the storage and network times, implying close overlap of the two phases (i.e., the shorter phase is effectively hidden). (4) Beyond runtime benefits, our overlapping approach also helps avoid OOM errors, as we explain in the next subsection. **Takeaway:** Overlapping storage and network I/O largely hides the shorter phase and significantly improves performance.

6.3 Dataset Scalability

Next, we study PystachIO’s performance behavior as the data size increases. Figure 16 reports the end-to-end runtimes of six representative TPC-H queries for scale factors from 100 to 1000. The main observations are: (1) PystachIO with all optimizations (green line) scales linearly with the SF and is close to the theoretical minimum. (2) The blocking variants run out of GPU memory as the dataset size increases (Q3, Q5, Q8, Q17), because their table scans attempt to materialize the full input in GPU memory. (3) For highly-selective queries with little shuffled data, such as Q14, the workload is primarily storage-bound and the networking and query processing phase is short, leading to both PystachIO and the optimized blocking variant being able to approach the minimum. (4) For queries with heavier shuffling, PystachIO benefits from our adaptive regulation technique (highlighted with a gray background), which limits the I/O queue size to avoid GPU OOM at the cost of slightly

slower storage throughput (see the slightly widening gap between PystachIO and optimum, e.g., for Q3 and Q5). (5) Opportunities for further optimizations become apparent for Q18, which suffers from operator overheads in the aggregation and large intermediate results. These situations motivate further research into more processing- and memory-efficient TCR-based query operators.

Takeaway: PystachIO is highly efficient and close to the theoretical minimum, due to overlapping techniques that improve runtime and reduce memory consumption, mitigating the OOM problems in large-scale data processing.

6.4 Cluster Scalability

We evaluate how PystachIO scales with the number of processing nodes using up to all 8 GPUs on the DGX system. We use the large scale factor SF1000, to determine at which cluster size the blocking variants overcome OOM failures. The results in Figure 18 show that: (1) Overlapping storage and network I/O, along with adaptive queue regulation, enables the execution of large datasets with significantly fewer nodes. The blocking variants require at least 8 nodes to fully materialize the inputs of Q5, Q8, and Q17, whereas PystachIO processes these queries efficiently with only 2 nodes by using adaptive regulation (which is not possible with the blocking variants). Only Q18 requires at least 6 GPUs for execution, due to the operator and memory overheads discussed before, leading to gaps in the optimal runtime. (2) PystachIO scales efficiently with an increasing number of GPU nodes, as the underlying storage and network primitives are highly scalable, and PystachIO can effectively orchestrate and overlap them. (3) Fully overlapped execution remains close to the theoretical minimum runtime as the number of nodes increases. Only Q18 shows the performance gap of large materialized aggregation, but the scaling to 8 nodes exhibits the expected decrease in runtime and decreases the size of the gap.

Takeaway: PystachIO scales efficiently with the number of nodes. By leveraging fast storage and overlapping I/O, it can process large datasets with a small number of nodes, enabling cost-efficient large-scale query processing.

6.5 Systems Comparison

Finally, we compare PystachIO against DuckDB [47] and ClickHouse [49], two CPU-based systems known for strong OLAP performance on storage-resident datasets. We also compare against Sirius [63], a GPU system that relies on CPU-based DuckDB for Parquet ingestion. We restrict this comparison to single-node query processing over SSD-resident data, because these systems do not support a distributed repartitioning join.

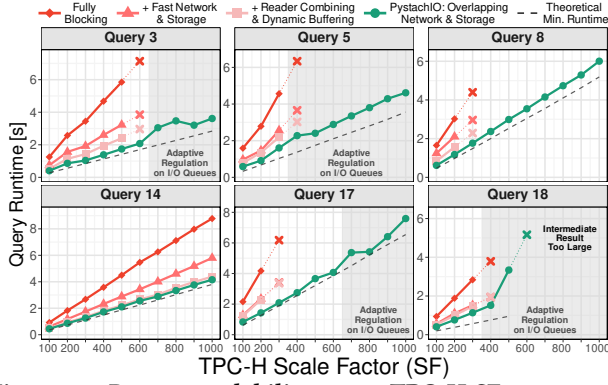


Figure 16: Dataset scalability up to TPC-H SF1000: query performance with 2 GPUs.

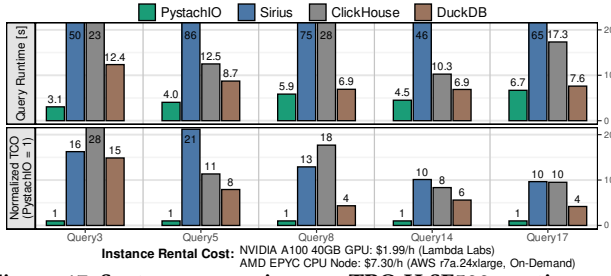


Figure 17: System comparison on TPC-H SF500 runtime and normalized TCO in a single-node setting with fully storage-resident data.

The runtimes in the upper part of Figure 17 show that PystachIO remains highly competitive even in this setting. DuckDB achieves similarly low runtimes for several queries, reflecting its efficient storage layer. However, PystachIO outperforms the other systems on queries with large scan inputs, such as Q3 and Q5, where its overlap of storage access and GPU execution is particularly effective. Sirius is also notable: although it often matches or exceeds PystachIO on in-memory workloads, it performs worse here because data loading and preparation remain CPU-driven, introducing substantial overhead for storage-resident execution. The lower part of Figure 17 converts these runtimes into total-cost-of-ownership (TCO) estimates using cloud VM prices for hardware comparable to our setup. Because PystachIO combines low query runtimes with competitive pricing for GPU-equipped instances, it achieves at least 4× lower TCO on the evaluated queries.

7 RELATED WORK

GPU Query Processing Many works [6, 22, 32, 33, 50] have proven the acceleration potential of GPUs for OLAP processing. Among them, the TQP series [7, 17, 19, 27] pioneered TCRs as execution engines and optimized their performance for in-memory scenarios. We adapt the TCR-based execution model for chunked execution and target I/O-heavy queries over fast networks and fast storage. **Extending GPU Query Processing Over Networks** Distributed GPU query processing has been studied in several systems [12, 13, 56, 63] that focus on GPU-resident or hybrid CPU-GPU memory workloads, do not use TCRs for execution, and do not consider storage. Theseus [3] explores distributed execution together with

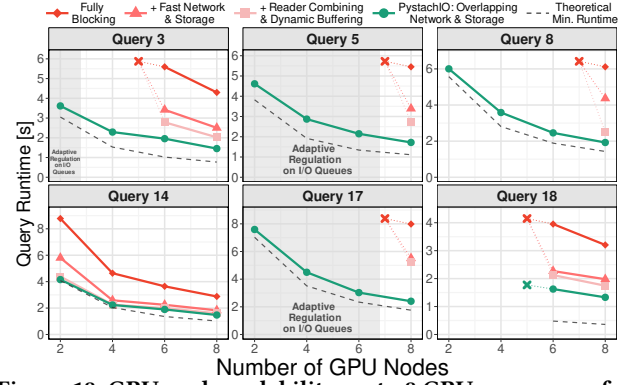


Figure 18: GPU-node scalability up to 8 GPUs: query performance with TPC-H SF1000.

data loading from SSDs; however, its system architecture has not yet been described in detail, which limits a thorough comparison with our approach. Wu et al. [61] deploy a TCR-based database on multi-node GPU clusters with NVLink and RDMA for communication, and demonstrate large-scale query execution. They (1) combine intra- and inter-node interconnects, (2) minimize network traffic by co-partitioning, and (3) assume that data is materialized in aggregate GPU memory. By contrast, we focus on distribution over RDMA networks, data loading from storage, and overlapping I/O with computation in TCRs. We expect their approach to approximate the networking performance of our blocking variants, since they process and shuffle data at column granularity.

Extending GPU Query Processing to Storage While GPU networking has been well studied, relatively few works have examined storage integration in GPU query processing, and none have considered TCRs. [5, 36] and [2, 18] study compression techniques and encodings for accelerated analytical processing, respectively. Although these approaches demonstrate the benefits of GPU-based decompression for specialized formats, industry systems [3, 23, 24, 41, 52, 64] increasingly focus on end-to-end query performance when loading data from open file formats such as Parquet. To our knowledge, we are the first to study fast storage access explicitly designed to overlap I/O with query processing in TCRs.

8 CONCLUSION

We presented PystachIO, a prototypical GPU-accelerated distributed analytical query engine built on the PyTorch TCR that exploits fast RDMA-capable networks and NVMe storage by overlapping storage & network I/O with GPU computation, achieving high throughput and low end-to-end query runtimes on large datasets.

ACKNOWLEDGMENTS

This research was partially funded by the state of Hesse as part of the NHR program (NHR4CES), the LOEWE Spitzenprofessur of the state of Hesse (III 5-519/05.00.003-(0005)), the project "hessian.AI AISC" (Grant No. 01IS22091) by the Federal Ministry of Education and Research (BMBF), as well as by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy (EXC3057/1 "Reasonable Artificial Intelligence", Project No. 533677015). We also thank DFKI Darmstadt and hessian.AI for their support.

REFERENCES

- [1] Daniel Abadi. 2025. Parquet and ORC’s many shortfalls for machine learning (ML) workloads, and what should be done about them. <https://www.starburst.io/blog/parquet-orc-machine-learning/>
- [2] Azim Afroozeh, Lotte Felius, and Peter Boncz. 2024. Accelerating GPU Data Processing using FastLanes Compression. In *Proceedings of the 20th International Workshop on Data Management on New Hardware, DaMoN 2024, Santiago, Chile, 10 June 2024*. ACM, 8:1–8:11. <https://doi.org/10.1145/3662010.3663450>
- [3] Felipe Aramburú, William Malpica, Kaouther Abrougui, Amin Aramoon, Romulo Auccapuclla, Claude Brisson, Matthijs Brobbel, Colby Farrell, Pradeep Garigipati, Joost Hoozemans, Supun Kamburugamuve, Akhil Nair, Alexander Ocsa, Johan Peltenburg, Rubén Quesada López, Deepak Sihag, Ahmet Uyar, Dhruv Vats, Michael Wendt, Jignesh M. Patel, and Rodrigo Aramburú. 2025. Theseus: A Distributed and Scalable GPU-Accelerated Query Processing Platform Optimized for Efficient Data Movement. *CoRR abs/2508.05029* (2025). <https://doi.org/10.48550/ARXIV.2508.05029> arXiv:2508.05029
- [4] Claude Barthels, Gustavo Alonso, Torsten Hoeftler, Timo Schneider, and Ingo Müller. 2017. Distributed Join Algorithms on Thousands of Cores. *Proc. VLDB Endow.* 10, 5 (2017), 517–528. <https://doi.org/10.14778/3055540.3055545>
- [5] Nils Boesch, Tobias Ziegler, and Carsten Binnig. 2024. GOLAP: A GPU-in-Data-Path Architecture for High-Speed OLAP. In *Proceedings of the 2025 International Conference on Management of Data, SIGMOD Conference 2025*. ACM, 237:1–237:26. <https://doi.org/10.1145/3698812>
- [6] Sebastian Breß, Henning Funke, and Jens Teubner. 2016. Robust Query Processing in Co-Processor-accelerated Databases. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016*. ACM, 1891–1906. <https://doi.org/10.1145/2882903.2882936>
- [7] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proc. VLDB Endow.* 17, 3 (2023), 441–454. <https://doi.org/10.14778/3632093.3632107>
- [8] NVIDIA Corporation. 2025. NVIDIA DGX A100 Hardware Overview. <https://docs.nvidia.com/dgx/dgxa100-user-guide/introduction-to-dgxa100.html>. Accessed: 2025-11-29.
- [9] Nuno Faria and Andrew Lamb with Apache DataFusion Contributors. 2025. DataFusion Pull Request: Cache Parquet metadata in built in parquet reader. <https://github.com/apache/datafusion/pull/16971>. Accessed: 2025-11-26.
- [10] NumFOCUS Foundation. 2025. NumPy. <https://numpy.org/>
- [11] NumFOCUS Foundation and Wes McKinney. 2025. pandas - Python Data Analysis Library. <https://pandas.pydata.org/>
- [12] Hao Gao and Nikolai Sakharov. 2021. Scaling Joins to a Thousand GPUs. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2021, Copenhagen, Denmark, August 16, 2021*. 55–64. http://www.adms-conf.org/2021-camera-ready/gao_adms21.pdf
- [13] Chengxin Guo, Hong Chen, Feng Zhang, and Cuiping Li. 2019. Distributed Join Algorithms on Multi-CPU Clusters with GPUDirect RDMA. In *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, Kyoto, Japan, August 05-08, 2019*. ACM, 65:1–65:10. <https://doi.org/10.1145/3337821.3337862>
- [14] Tushar Gupta. 2024. Evolution of Data Center Design to Handle AI Workloads. In *34th International Telecommunication Networks and Applications Conference, ITNAC 2024, Sydney, Australia, November 27-29, 2024*. IEEE, 1–8. <https://doi.org/10.1109/ITNAC62915.2024.10815309>
- [15] Xiangpeng Hao. 2024. Caching in DataFusion. <https://blog.xiangpeng.systems/posts/caching-datafusion/>. Accessed: 2025-11-26.
- [16] Xiangpeng Hao and Andrew Lamb. 2024. How Good is Parquet for Wide Tables (Machine Learning Workloads) Really? <https://www.influxdata.com/blog/how-good-parquet-wide-tables/>
- [17] Dong He, Supun Chathuranga Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query Processing on Tensor Computation Runtimes. *Proc. VLDB Endow.* 15, 11 (2022), 2811–2825. <https://doi.org/10.14778/3551793.3551833>
- [18] Sven Hepkema, Azim Afroozeh, Charlotte Felius, Peter Boncz, and Stefan Manegold. 2025. G-ALP: Rethinking Light-weight Encodings for GPUs. In *Proceedings of the 21st International Workshop on Data Management on New Hardware, DaMoN 2025, Berlin, Germany, June 22-27, 2025*. ACM, 11:1–11:10. <https://doi.org/10.1145/3736227.3736242>
- [19] Zehou Huang, Krystian Sakowski, Hans Lehnert, Wei Cui, Carlo Curino, Matteo Interlandi, Marius Dumitru, and Rathijit Sen. 2025. GPU Acceleration of SQL Analytics on Compressed Data. *Proc. VLDB Endow.* 19, 3 (2025), 320–333. <https://www.vldb.org/pvldb/vol19/p320-sen.pdf>
- [20] Matthias Jasny, Muhammad El-Hindi, Tobias Ziegler, and Carsten Binnig. 2025. A Wake-Up Call for Kernel-Bypass on Modern Hardware. In *Proceedings of the 21st International Workshop on Data Management on New Hardware, DaMoN 2025, Berlin, Germany, June 22-27, 2025*. ACM, 14:1–14:5. <https://doi.org/10.1145/3736227.3736235>
- [21] Matthias Jasny, Muhammad El-Hindi, Tobias Ziegler, Viktor Leis, and Carsten Binnig. 2025. High-Performance DBMSs with io_uring: When and How to use it. *CoRR abs/2512.04859* (2025). <https://doi.org/10.48550/ARXIV.2512.04859> arXiv:2512.04859
- [22] Marko Kabic, Shriram Chandran, and Gustavo Alonso. 2025. Maximus: A Modular Accelerated Query Engine for Data Analytics on Heterogeneous Systems. In *Proceedings of the 2025 International Conference on Management of Data, SIGMOD Conference 2025*. ACM, 187:1–187:25. <https://doi.org/10.1145/3725324>
- [23] Greg Kimball, Zoltan Arnold Nagy, Devavret Makkar, Daniel Bauer, and Chengcheng Jin. 2025. Accelerating Large-Scale Data Analytics with GPU-Native Velox and NVIDIA cuDF. https://developer.nvidia.com/blog/accelerating-large-scale-data-analytics-with-gpu-native-velox-and-nvidia-cudf/#hybrid_cpu-gpu_execution_in_apache_spark Accessed: 2025-11-26.
- [24] Greg Kimball and Karthikeyan Natarajan. 2025. Accelerating Velox with RAPIDS cuDF - VeloxCon 2025. <https://prestodb.io/wp-content/uploads/presto-users/Accelerating-Velox-with-RAPIDS-cuDF-VeloxCon-April-2025.pdf> Accessed: 2025-11-26.
- [25] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. In *Proceedings of the 2023 International Conference on Management of Data, SIGMOD Conference 2023*. ACM, 118:1–118:26. <https://doi.org/10.1145/3589263>
- [26] Andrew Lamb. 2025. Apache Parquet Metadata Parsing Benchmarks. https://github.com/alamb/parquet_footer_parsing. Accessed: 2025-11-26.
- [27] Yinan Li, Bailu Ding, Ziyun Wei, Lukas M. Maas, Momin Al-Ghousien, Spyros Blanas, Nicolas Bruno, Carlo Curino, Matteo Interlandi, Craig Peeper, Kaushik Rajan, Surajit Chaudhuri, and Johannes Gehrke. 2025. Scaling GPU-Accelerated Databases beyond GPU Memory Size. *Proc. VLDB Endow.* 18, 11 (2025), 4518–4531. <https://doi.org/10.14778/3749646.3749710>
- [28] Jigao Luo. 2025. cuDF Issue: Unstable pipelining performance in Parquet reading due to mis-sync. <https://github.com/rapidsai/cudf/issues/18967>. Accessed: 2025-11-26.
- [29] Jigao Luo. 2025. [Story] Towards a faster Parquet reader with pipelining and multistream optimization. <https://github.com/rapidsai/cudf/issues/18892>. Accessed: 2025-11-27.
- [30] Jigao Luo, Qi Chen, and Carsten Binnig. 2026. Do GPUs Really Need New Tabular File Formats? *CoRR abs/2602.17335* (2026). <https://doi.org/10.48550/ARXIV.2602.17335> arXiv:2602.17335
- [31] Jigao Luo and Muhammad Haseeb. 2025. cuDF POC Pull Request: Metadata caching prototype in Parquet reader. <https://github.com/rapidsai/cudf/pull/18891>. Accessed: 2025-11-26.
- [32] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020*. ACM, 1633–1649. <https://doi.org/10.1145/3318464.3389705>
- [33] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects. In *Proceedings of the 2022 International Conference on Management of Data, SIGMOD Conference 2022*. ACM, 1017–1032. <https://doi.org/10.1145/3514221.3517911>
- [34] Linux Foundation Meta AI. 2025. PyTorch. <https://pytorch.org/>
- [35] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. <https://doi.org/10.14778/2002938.2002940>
- [36] Hamish Nicholson, Konstantinos Chasialis, Antonio Boffa, and Anastasia Ailamaki. 2025. The Effectiveness of Compression for GPU-Accelerated Queries on Out-of-Memory Datasets. In *Proceedings of the 21st International Workshop on Data Management on New Hardware, DaMoN 2025, Berlin, Germany, June 22-27, 2025*. ACM, 10:1–10:10. <https://doi.org/10.1145/3736227.3736240>
- [37] Jesse Noffsinger, Maria Goodpaster, Mark Patel, Haley Chang, Pankaj Sachdeva, and Arjita Bhan. 2025. The cost of compute: A \$ 7 trillion race to scale data centers. <https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/the-cost-of-compute-a-7-trillion-dollar-race-to-scale-data-centers>
- [38] NVIDIA. 2025. NCCL. <https://developer.nvidia.com/nccl>
- [39] NVIDIA. 2025. NVIDIA GPUDirect RDMA. <https://docs.nvidia.com/cuda/gpudirect-rdma/>
- [40] NVIDIA. 2025. NVIDIA GPUDirect Storage. <https://docs.nvidia.com/gpudirect-storage/>
- [41] NVIDIA. 2025. RAPIDS Accelerator for Apache Spark. <https://docs.nvidia.com/spark-rapids/index.html>. Accessed: 2025-11-28.
- [42] NVIDIA Corporation. 2025. CUB: Cooperative primitives for CUDA. <https://nvidia.github.io/ccl/cub/> Accessed: 2025-11-26.
- [43] NVIDIA Corporation. 2025. CUDA C Programming Guide: Concurrent Execution between Host and Device. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#concurrent-execution-host-device> Accessed: 2025-11-26.
- [44] NVIDIA Corporation. 2025. CUDA Runtime API: API Synchronization Behavior. https://docs.nvidia.com/cuda/cuda-runtime-api/api-sync-behavior.html#api-sync-behavior__memcpy-async Accessed: 2025-11-26.

- [45] NVIDIA Corporation. 2025. *Thrust: Parallel Algorithms Library*. <https://developer.nvidia.com/thrust> Accessed: 2025-11-26.
- [46] Weston Pace, Chang She, Lei Xu, Will Jones, Albert Lockett, Jun Wang, and Raunak Shah. 2025. Lance: Efficient Random Access in Columnar Storage through Adaptive Structural Encodings. *CoRR* abs/2504.15247 (2025). <https://doi.org/10.48550/ARXIV.2504.15247> arXiv:2504.15247
- [47] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019*. ACM, 1981–1984. <https://doi.org/10.1145/3299869.3320212>
- [48] NVIDIA NCCL Repository. 2025. NCCL Overlap Issue. <https://github.com/NVIDIA/nccl/issues/338>
- [49] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse - Lightning Fast Analytics for Everyone. *Proc. VLDB Endow.* 17, 12 (2024), 3731–3744. <https://doi.org/10.14778/3685800.3685802>
- [50] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020*. ACM, 1617–1632. <https://doi.org/10.1145/3318464.3380595>
- [51] Amazon Staff. 2025. Amazon to invest up to \$ 50 billion to expand AI and supercomputing infrastructure for US Government agencies. <https://www.aboutamazon.com/news/company-news/amazon-ai-investment-us-federal-agencies>
- [52] Polars Team. 2024. GPU acceleration with Polars and NVIDIA RAPIDS. <https://polars.rs/posts/gpu-engine-release/>. Accessed: 2025-11-28.
- [53] RAPIDS Development Team. 2025. NVIDIA RAPIDS KvikIO. <https://docs.rapids.ai/api/kvikio/stable/>
- [54] RAPIDS Development Team. 2025. NVIDIA RAPIDS libcudf, pylibcudf and cudf: GPU DataFrame Library. <https://github.com/rapidsai/cudf>
- [55] RAPIDS Development Team. 2025. NVIDIA RMM: RAPIDS Memory Manager. <https://github.com/rapidsai/rmm>
- [56] Lasse Thostrup, Gloria Doci, Nils Boeschen, Manisha Luthra, and Carsten Binnig. 2023. Distributed GPU Joins on Fast RDMA-capable Networks. In *Proceedings of the 2023 International Conference on Management of Data, SIGMOD Conference 2023*. ACM, 29:1–29:26. <https://doi.org/10.1145/3588709>
- [57] Deepak Vohra. 2016. *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools* (1st ed.). Apress, USA. <https://doi.org/10.1007/978-1-4842-2199-0>
- [58] Yunsong Wang. 2025. cuCollections Pull Request: Use pinned host memory for improved async performance. <https://github.com/NVIDIA/cuCollections/pull/727>. Accessed: 2025-11-27.
- [59] Jake Hemstad with NVIDIA CCCL Team. 2023. CCCL Issue: Universal 64-bit index type support in Thrust/CUB algorithms. <https://github.com/NVIDIA/cccl/issues/47>. Accessed: 2025-11-26.
- [60] Greg Kimball with RAPIDS cudf Team. 2023. cudf Issue: Add 64-bit size type option at build-time for libcudf. <https://github.com/rapidsai/cudf/issues/13159>. Accessed: 2025-11-26.
- [61] Bowen Wu, Wei Cui, Carlo Curino, Matteo Interlandi, and Rathijit Sen. 2025. Terabyte-Scale Analytics in the Blink of an Eye. *Proc. VLDB Endow.* 19, 2 (2025), 141–155. <https://www.vldb.org/pvldb/vol19/p141-sen.pdf>
- [62] Zhisheng Ye, Wei Gao, Qinghao Hu, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. 2024. Deep Learning Workload Scheduling in GPU Datacenters: A Survey. *ACM Comput. Surv.* 56, 6 (2024), 146:1–146:38. <https://doi.org/10.1145/3638757>
- [63] Bobbi Yogatama, Yifei Yang, Kevin Kristensen, Devesh Sarda, Abigale Kim, Adrian Cockcroft, Yu Teng, Joshua Patterson, Gregory Kimball, Wes McKinney, Weiwei Gong, and Xiangyao Yu. 2026. Rethinking Analytical Processing in the GPU Era. In *16th Conference on Innovative Data Systems Research, CIDR 2026, Chaminade, CA, USA, January 18-21, 2026*. www.cidrdb.org. <https://vldb.org/cidrdb/2026/rethinking-analytical-processing-in-the-gpu-era.html>
- [64] Ben Zaitlen, Peter Entschew, and Rick Zamora. 2024. Best Practices for Multi-GPU Data Analysis Using RAPIDS with Dask. <https://developer.nvidia.com/blog/best-practices-for-multi-gpu-data-analysis-using-rapids-with-dask/>. Accessed: 2025-11-28.
- [65] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *Proceedings of the 2022 International Conference on Management of Data, SIGMOD Conference 2022*. ACM, 685–699. <https://doi.org/10.1145/3514221.3526187>