The Effectiveness of Compression for GPU-Accelerated Queries on Out-of-Memory Datasets

Hamish Nicholson
hamish.nicholson@epfl.ch
EPFL
Lausanne, Switzerland

Konstantinos Chasialis*
konstantinos.chasialis@oracle.com
Oracle
Zurich, Switzerland

Antonio Boffa
antonio.boffa@epfl.ch
EPFL
Lausanne, Switzerland

Anastasia Ailamaki
anastasia.ailamaki@epfl.ch
EPFL
Lausanne, Switzerland

Abstract

GPU-accelerated database systems promise significant performance gains for analytical workloads, but their adoption remains limited due to fundamental data movement bottlenecks when the dataset does not fit in GPU memory. Compression can significantly accelerate data movement, but when the dataset does not fit in GPU memory, the choice of compression algorithm (lightweight or heavyweight) can have a wide range of effects, including a potential slowdown compared to uncompressed data.

This paper analyzes the impact of compression on query execution performance for larger-than-GPU-memory datasets stored on arrays of NVMe SSDs. We integrate pipelined data movement and on-GPU decompression into a GPU query processing engine. Our experiments show that when storage bandwidth is limited, compression always accelerates query execution, with lightweight compression incurring a higher speedup than heavyweight compression. As storage bandwidth is increased, however, compression can incur a *speedup* or a *slowdown*, depending on the processing intensity of the query. We quantify the impact of compression on query execution time with a microbenchmark that varies both storage bandwidth as well as the selectivity and number of joins. We find that heavyweight compression suffers from performance degradation with less processing-intensive queries and lower storage bandwidths than lightweight compression.

We further analyze the compression algorithm's impact on GPU utilization using an extended roofline model incorporating the SSD-to-GPU bandwidth bottleneck. For queries from the Star Schema Benchmark, the results show that while lightweight compression drastically increases the count of executed instructions, the improvement in effective storage bandwidth results in improved query execution times.

*Work done entirely while the author was at EPFL.

DaMoN '25, June 22–27, 2025, Berlin, Germany

ACM ISBN 979-8-4007-1940-0/2025/06
https://doi.org/10.1145/3736227.3736240

CCS Concepts

• **Information systems** → *Online analytical processing engines*; **Data scans**; **Database query processing**; *Flash memory*.

Keywords

GPU acceleration, Data compression, Analytical query processing, NVMe SSD, OLAP workloads

ACM Reference Format:
Hamish Nicholson, Konstantinos Chasialis, Antonio Boffa, and Anastasia Ailamaki. 2025. The Effectiveness of Compression for GPU-Accelerated Queries on Out-of-Memory Datasets. In *21st International Workshop on Data Management on New Hardware (DaMoN '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3736227.3736240

1 Introduction

Using graphics processing units (GPUs) to accelerate analytical query processing offers significant performance improvements when the dataset fits in GPU memory [8, 20, 41]. Nevertheless, when data is not already in GPU memory, data movement is the primary query execution bottleneck, limiting the adoption of GPU-enabled DBMS due to constraints of the interconnect bandwidth [19, 28, 35, 39].

Despite the inherent interconnect limitations, using arrays of PCIe-attached NVMe drives is a promising approach for cost-effective query processing with large data sets. Arrays of NVMe drives now offer 10s-100s of GB/s of storage bandwidth, substantially closing the performance gap between DRAM and storage [22, 23]. Recent work has shown to exploit arrays of NVMe drives as a cost-effective solution to achieve near-in-memory performance with larger-than-memory data in CPU-based data management systems [23, 24, 32, 33]. However, unlike modern CPUs, which have up to 128 PCIe lanes [3], GPUs typically have a maximum of 16 PCIe lanes and, therefore, cannot benefit from the same amount of raw storage bandwidth as CPUs. Consequently, GPU-based query processing requires techniques different from CPU-based systems to benefit from fast and efficient NVMe storage.

An intuitive approach to leverage NVMe storage for GPUs is data compression. By reducing the volume of data that is moved, the available bandwidth can be better utilized [10, 18]. Further, as real-world datasets become increasingly redundant [31, 44, 49],

compression techniques can achieve even greater compression ratios and therefore higher effective data movement bandwidths. By storing data compressed before movement and decompressing it on the fly, systems can reduce their storage footprint and alleviate pressure on interconnect bottlenecks.

Following preliminary results on moving data compressed to the GPU [26, 48], research has shifted towards in-GPU-memory processing [7, 12, 41, 42], targeting a use case that assumes the dataset is small enough to fit in GPU memory and has already been loaded (or cached). To achieve high throughput, such approaches often introduce new lightweight encoders and rely on manually optimized GPU kernels tailored to specific operator pipelines, maximizing performance within the constraints of in-memory execution. This raises the question of how compression influences the performance of systems that load data to the GPU during query execution in order to operate on data that exceeds the GPU memory capacity. Recently, Boeschen et al.[10] proposed a GPU-in-data-path architecture to accelerate table scans by streaming data compressed with heavyweight compression directly from SSDs to the GPU, where it is decompressed on the fly as part of a table scan. Previous work has introduced new lightweight compression for in-GPU-memory processing and leveraged heavyweight compression for GPU-NVMe systems. The design space of compression for GPU query processing, especially in larger-than-GPU-memory scenarios, is complex, with interactions and interference between decompression and query processing itself.

Contributions. We conduct experiments to answer the following questions: (1) How does the choice of compression algorithm (lightweight vs heavyweight) impact the performance of GPU-accelerated analytical workloads? (2) Which query parameters determine the performance of compression algorithms? (3) How does streaming decompression during query execution affect GPU utilization?

We extend a GPU query processing engine to pipeline and overlap, data movement, decompression, and query processing. We evaluate this system along three axes: (1) NVMe-to-GPU bandwidth; (2) data compression, where we assess GPU-optimized heavyweight and lightweight compression; and (3) query characteristics such as selectivity and number of joins that influence the query processing intensity. We analyze performance using a larger-than-GPU-memory dataset and assume a cold cache which requires moving data from NVMe storage to the GPU during query execution.

Our experimental study leads to three key findings that will guide future practitioners and researchers working on GPU-accelerated query processing systems:

(1) We identify the key factors that govern whether compression improves or degrades query performance: storage bandwidth, query complexity (e.g., number of joins), and selectivity. Specifically, we find that beyond four NVMe drives, in queries with more than three joins, and when selectivity exceeds 50%, decompression begins to compete with query execution for GPU resources. In these cases, even efficient lightweight compression can underperform compared to uncompressed execution (Section 4.1).
(2) We demonstrate that compression consistently improves query execution performance across all tested SSB workloads when the system is storage-bound (e.g., using a single NVMe drive), with speedup factors closely aligning with compression ratios. However, as storage bandwidth increases, heavyweight compression schemes may degrade performance due to their higher decompression overhead. By contrast, lightweight compression continues to deliver speedups even under high-bandwidth conditions (Section 4.2).
(3) We apply a modified roofline model to analyze the storage-boundedness of SSB queries. Unsurprisingly, we find that query execution with uncompressed data underutilizes the GPU resources and is heavily storage-bound. However, we also find that while query execution with lightweight decompression results in a drastic increase, by up to 28.6× more, in the number of instructions executed, query execution time still improves due to the reduced data movement to the GPU (Section 4.3).

The rest of the paper is organized as follows: section 2 introduces the background on compression algorithms and the trade-offs for efficient decompression for GPU query processing. Section 3 provides an overview of the system we use for analysis and how we integrate pipelined decompression with query execution. Our experimental results follow in section 4 with related work and conclusions in section 5 and section 6, respectively.

2 Compression in GPU-Accelerated Analytics

GPU databases utilize custom lightweight and heavyweight compression techniques. Due to GPU memory capacity constraints when processing analytical queries, pipelining and streaming data movement and decompression are necessary to alleviate the ensuing bandwidth bottleneck. This section introduces the related algorithms and tradeoffs for efficient query processing.

2.1 Lightweight vs. Heavyweight Compression

Lightweight encoding schemes, such as run-length encoding (RLE), dictionary encoding [9], delta encoding, packing, bitmaps [13], FSST [11], XOR [37], Null suppression [40], and learned schemes [21, 27], are designed to minimize computational overhead. These methods can perform exceptionally well on specific types of data, such as strings, integers, and floating-point values, especially when the data exhibit patterns like high repetitiveness and serial correlation or are sorted or clustered. Their simplicity allows for the possibility of query execution while decoding the data (kernel fusion [42]), further reducing latency and computational costs.

Cascaded [48] encoding applies multiple lightweight compression techniques. It combines bit-packing, RLE, and delta encoding, leveraging the strengths of each approach. Nvidia's implementation [2] is optimized for GPUs: bit-packing aligns values within machine words, RLE and delta encoding use efficient prefix sum operations from the CUB [1] library.

Heavyweight encoding techniques also referred to as general-purpose compressors (the most used ones are based on Lempel-Ziv [25] compression), are designed to achieve high compression ratios by identifying and exploiting long repetitions in the input data, which are stored as match copies (length + distance). By analyzing large chunks of data, heavyweight schemes achieve high compression on data with high redundancy. However, this comes at

the cost of increased computational overhead, as they often require full decompression before query execution, which can introduce additional latency.

DEFLATE applies Huffman entropy coding [29] to further improve compression ratios; this additional entropy coding step makes it particularly effective for datasets with high redundancy but also slower. To enhance parallel decompression for GPUs, GDeflate [2] optimizes DEFLATE [16] by restructuring the bitstream into 32 independent sub-streams, enabling 32-way parallelism during decompression. It eliminates data dependencies by ensuring that match copies (length + distance) remain within the same sub-stream, avoiding cross-thread synchronization. In contrast, LZ4 is a byte-oriented compression scheme that does not use entropy coding, generally resulting in lower compression ratios than GDeflate but with faster encoding/decoding. To efficiently parallelize LZ4, the data is divided into a series of independent blocks, with each block compressed or decompressed concurrently.

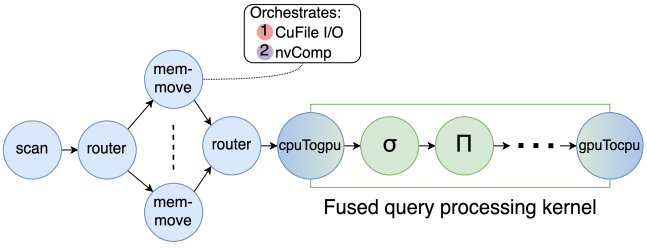
2.2 Pipelined Decompression

Due to limited GPU memory capacity, streaming decompression, i.e., decompressing smaller units as they are moved to the GPU rather than moving then decompressing entire columns or tables at a time, is necessary for GPU query execution over larger-than-GPU-memory datasets. For example, the uncompressed size of the lineorder columns scanned by SSB Q1.1-3 at scale factor 1000 is 96 GB while the compressed size with Cascaded compression is 36.5 GB (see Table 1). Even though the compressed columns will just fit in the 48 GB of an Nvidia A40 or more comfortably in the 80 GB of an A100, there is not sufficient memory capacity to allocate memory for the decompressed columns.

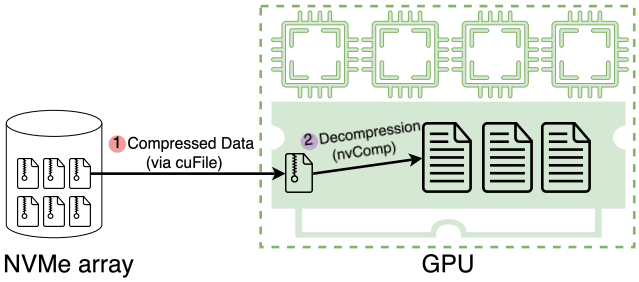
Query	Raw GB	LZ4 GB (ratio)	GDeflate GB (ratio)	Cascaded GB (ratio)
Q1.1	96	65.7 (1.46)	49.0 (1.96)	36.5 (2.63)
Q2.1	96	79.9 (1.20)	75.0 (1.28)	56.5 (1.70)
Q3.1	96	65.5 (1.47)	77.0 (1.25)	45.4 (2.11)
Q4.1	144	110.8 (1.30)	112.7 (1.28)	75.4 (1.91)
Q*J	144	107.3 (1.34)	103.6 (1.39)	69.4 (2.07)

Table 1: Uncompressed and compressed data sizes (in GB) and compression ratio using different compression schemes for SSB queries. Cascaded always achieves the lowest compressed data size, and therefore the highest compression ratio.

Further, for performance data movement, decompression and query processing should overlap, i.e., be pipelined. The data movement bottleneck is exacerbated if data movement and decompression occur serially with respect to the execution of the query operators. Following the previous SSB example, unpipelined data movement/compression will result in the query operators stalled for 1.1s, even assuming decompression is instantaneous (and assuming the theoretical PCIe 4.0 16x bandwidth of 31.5 GB/s, the practical bandwidth upper bound is lower.)



(a) Control flow



(b) Data flow

Figure 1: Query processing pipeline in Proteus extended with pipelined decompression. (a) Control flow with operators executing on the CPU in blue, and on the GPU in green. (b) The data flow illustrating the direct movement of compressed row groups to the GPU and decompression on the GPU. For each row-group, the mem-move operator makes the CPU-side API calls that initiate both operations.

3 System Design

We conduct experiments using Proteus [14, 15, 17, 35, 39], a code-generating query processing engine with support for query execution on CPU/GPU platforms. We extend the engine with pipelined decompression for compressed table scans.

Proteus utilizes a push-based processing model and a columnar data format. Data is moved in row groups, each composed of a vector of values for each scanned column. The maximum vector size within a row group is 2 MiB. Relational operators are code-generated, and a pipeline of relational operators is fused into a single kernel. At runtime, a cpuTogpu operator executing on the CPU transfers control flow to the GPU by launching the GPU operator pipeline kernel. The arguments for each operator pipeline kernel launch include pointers to GPU memory for a row group. So, for a table scan, the operator pipeline kernel is invoked once for each row group, with the pipeline state maintained in memory across kernel invocations. Within a GPU operator pipeline kernel, each thread operates on a tuple at a time. On Nvidia GPUs the default number of thread blocks for kernel launches is twice the number of streaming multiprocessors with 1024 threads per thread block.

Figure 1a presents a physical plan incorporating pipelined decompression, illustrating the control flow between CPU and GPU while fig. 1b shows the accompanying data flow during query execution. For ease of illustration, fig. 1 depicts a single pipeline physical plan without pipeline breakers, such as hash joins. The scan operator

emits a stream of records containing the metadata necessary to load a row group from storage. Data movement to the GPU is performed by the mem-move operator, which executes on the CPU. The mem-move operator initiates I/O using the synchronous cuFile (version 2.12) API using NVIDIA's GPU Direct Storage (GDS) technology [6] to move data directly from NVMe drives into GPU memory without first going through a CPU memory buffer (1 in fig. 1). We use the synchronous API because we were unable to saturate the GPU interconnect bandwidth with the asynchronous API, a result replicated by other papers [10]. We extend the mem-move operator to also perform decompression using the asynchronous nvCOMP (version 4.2) APIs to decompress the data, which internally launches the decompression kernels on the GPU, overlapping data movement and decompression within each mem-move operator instance and writing the decompressed data back to global memory (2 in fig. 1). In order to saturate the interconnect bandwidth when using multiple NVMe drives, we use 16 parallel mem-move instances. An initial router operator [15] after the scan operator routes row group metadata records to the mem-move instances. A router maintains an asynchronous queue for each consumer. A second router is placed between the mem-move instances and the cpu2gpu operator instance. This router adapts from the CPU operator degree of parallelism (16) to the GPU operator degree of parallelism (the number of GPUs), and routes records containing the pointers to GPU memory for the row group. The router's asynchronous queue results in pipelined and overlapping decompression and relational operator pipeline kernel execution on the GPU.

4 Experimental Evaluation

Hardware. Our evaluation is conducted on a server with a 2x24-core AMD EPYC 7413 processor, having two threads per core, totalling 96 threads and 256 GB of DRAM. Each CPU socket is connected with a single Nvidia A40 GPU with 48 GB GDDR6 memory using 16 PCIe 4.0 lanes and 12 Corsair MP600 Pro NVMe drives, each using 4 PCIe 4.0 lanes. The theoretical maximum bandwidth of the GPU interconnect is 31.5 GB/s. We observe 86 GB/s sequential read bandwidth from all 12 NVMe to system DRAM when using `fio`. All experiments were conducted on a single socket and utilized all 12 NVMe drives unless otherwise stated. The GPU interconnect bandwidth is saturated when using between 4 and 8 NVMe drives.
Compression Algorithms. We evaluate two heavyweight compression algorithms, LZ4, and GDeflate [2, 16], as well as the lightweight NVIDIA nvComp Cascaded [2]. Decompression is performed entirely on the GPU using nvCOMP (version 4.2). For all compression algorithms, we use a chunk size of 64 KiB.
Dataset. We use the Star Schema Benchmark (SSB) [36] queries and dataset at scale factor 1000. For string columns, regardless of the compression scheme used, we apply dictionary encoding during data preparation. Each unique string value is mapped to a 4-byte integer ID using a static ordered dictionary constructed from the full column. Dictionaries are stored on the NVMe drives and loaded into system DRAM during query execution. This transformation reduces both data size and decoding overhead during query execution. The resulting integer values are stored on the SSD drives and treated as regular integer columns, supporting equality comparisons and range predicates. More complex operations, such as general "LIKE" pattern matching, are not supported.

Without compression each lineorder column is 24 GB. SSB queries consist of four groups, within each group, the query selectivity decreases with the rank; for example, Q1.3 is more selective than Q1.2, which is more selective than Q1.1. Query groups 1-3 scan 96 GB and group 4 scans 144 GB per query (see Table 1). In all query groups, the scan data volume is dominated by the lineorder columns.

4.1 Sensitivity to Query Characteristics

We evaluate the execution time of a series of microbenchmark queries using no compression, Cascaded, LZ4 and GDeflate for varying storage bandwidths by striping the data across different numbers of NVMe drives. The 5 queries (Q0J, Q1J, Q2J, Q3J, Q4J) are each over the same 6 columns from the lineorder table of the SSB: the four foreign key columns and `lo_revenue` and `lo_quantity`. The queries differ by how many dimension tables are joined with the lineorder table. Q0J performs no joins, Q1J join with date, Q2J joins with date and supplier and so on for the remaining queries joining with the other dimension tables in order of dimension table size. For these queries, Proteus employs a left-deep query plan with hash joins, where a hash table is built on each dimension table, and the subsequent probe phases are pipelined across all joins. We vary the selectivity of the queries with a filter on `lo_quantity` before the joins. A higher selectivity results in more random accesses in the probe phase of the hash joins. No filter is applied to the dimension tables. Listing 1 shows the SQL for query Q4J. The uncompressed data scanned by each query is 144 GB. The compressed size of the data depends on the compression algorithm used: LZ4 107.3 GB, Cascaded, 69.4 GB, GDeflate 103.6 GB (see Table 1). This workload enables us to quantify the benefit and potential interference between compression and query processing because the data moved to the GPU is essentially the same for all queries, as the dimension columns are at most, in Q4J, 0.19% of the data volume scanned.

```
SELECT SUM(lo_revenue)
FROM date, customer, supplier, part, lineorder
WHERE lo_custkey = c_custkey
  AND lo_suppkey = s_suppkey
  AND lo_partkey = p_partkey
  AND lo_orderdate = d_datekey
  AND lo_quantity < {FILTER_SELECTIVITY_PARAM};
```

Listing 1: Microbenchmark query Q4J: Lineorder fact table joined with four dimension tables.

Figure 2 plots the results of these microbenchmarks. Without compression, execution time scales proportionally to the storage bandwidth until the interconnect bottleneck is reached between 4 and 8 drives. For all compression algorithms, the execution time is further influenced by the selectivity and number of joins, as both parameters control the amount of work the query processing kernels perform. When using 8 drives, with increasing selectivity there is a clear inflection point where execution time with decompression begins to decrease. With more joins, the inflection point begins at lower selectivity. The inflection point is also at lower selectivities with the heavyweight compression algorithms (LZ4 and GDeflate) than with the lightweight Cascaded.

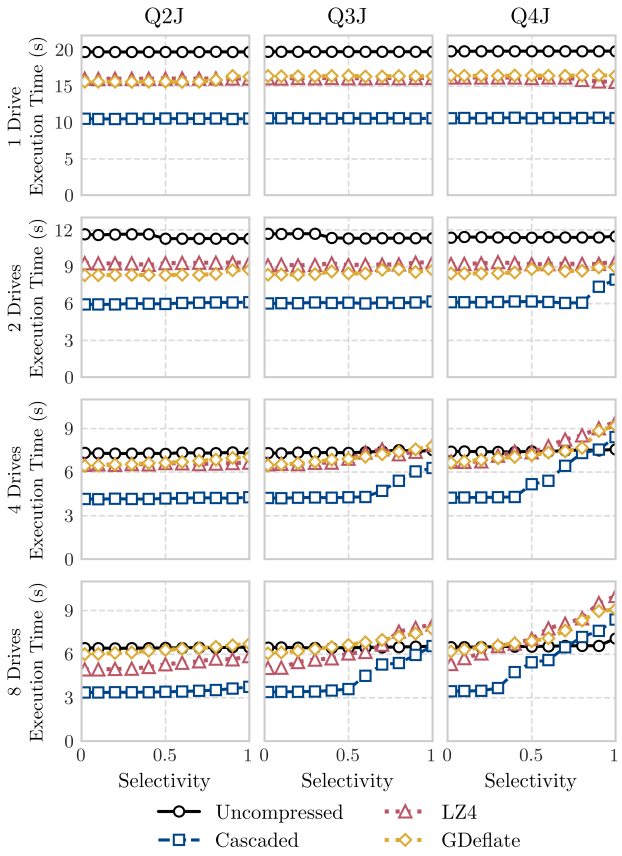


Figure 2: Query execution time against fact table selectivity by compression algorithm compared to no compression. Each row uses an increasing number of drives, and therefore has greater storage bandwidth. Between 4 and 8 drives, the storage bandwidth exceeds the GPU interconnect bandwidth.

Lightweight Cascaded compression performs better than the heavyweight GDeflate and LZ4, because it is tailored for columnar datasets, where values often exhibit sequential dependencies. This allows Cascaded to achieve higher data reduction while keeping decompression efficient. Using one drive at selectivity 0.02 across all queries, LZ4 achieves a 1.22×-1.23× speedup relative to uncompressed, GDeflate a 1.20×-1.26× speedup and Cascaded a 1.78× to 1.98× speedup.

At lower storage bandwidths, the small compression ratio advantage of GDeflate (LZ4 1.34, GDeflate 1.39) helps mitigate the data transfer bottleneck, leading to similar execution times between the two schemes. However, as storage bandwidth increases, the impact of compression ratio diminishes, and decompression speed becomes the dominant factor. At higher bandwidths, LZ4 achieves a lower execution time than GDeflate because it is a byte-oriented compression scheme that prioritizes fast decompression.

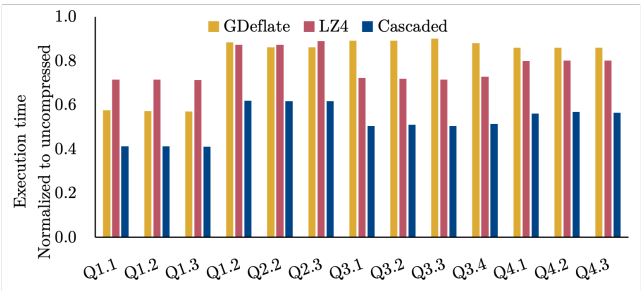
Discussion. Query execution times with compressed data do not depend solely on the compression ratio. The decompression kernels interfere with the query processing kernels, resulting in query processing kernels taking longer to perform the same work. The magnitude of the performance impact depends on the amount of work the query processing kernel performs, as higher selectivities and more joins lead to increased hash table probes. To quantify this, we trace the time spent launching and waiting on each GPU query processing kernel invocation (i.e., not the decompression kernels) in the cpuTogpu operator within the Q4J pipeline that scans the lineorder table and probes the dimension hash tables.

Using 8 NVMe drives, our measurements for Q4J show that at a low selectivity (0.02), the query processing kernels take 0.87s without compression, but increase to 1.16s with Cascaded and 5.8s with LZ4 due to this interference. At full selectivity (1.0), the baseline kernel time rises to 6.6s, and with compression, it further increases to 7.8s for Cascaded and 9.8s for LZ4, demonstrating that the overhead from interference scales with the underlying query processing workload. Notably, at low selectivity, these traced probe-side kernel times are significantly less than the overall query execution time, primarily because the pipeline is frequently bottlenecked by data movement and decompression, in addition to the 180- 280ms for the dimension table build phase, which is not included in the trace.

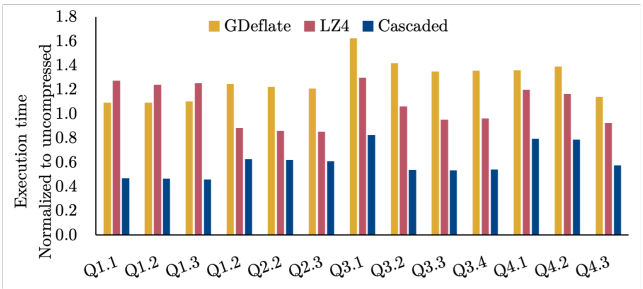
To understand how compression interferes with query processing, we use the CUPTI PC Sampling API. This allows us to profile concurrently running kernels with low overhead. At a fixed interval of cycles, CUPTI picks a random active warp on the streaming multi-processor (SM) and records its scheduler state. The scheduler state records if and why this warp did not dispatch an instruction, and if any warp on the SM dispatched an instruction.

The overall SM activity, measured by the proportion of samples where at least one instruction was dispatched by any warp on the SM, increases with compression. For instance, at 0.02 selectivity, this issue ratio rises from 53.5% (no compression) to 58.1% (Cascaded) and 61.4% (LZ4); a similar trend is observed at selectivity 1.0 (from 52.1% for no compression to 54.5% for Cascaded and 57.6% for LZ4, respectively). This indicates that the additional active decompression kernels contribute to overall SM instruction throughput. However, decompression also results in more overall instructions and warps being executed, as well as an increase in stalled warps between low and high selectivity due to increased contention.

Cascaded decompression leads to a sharp increase in warp stalls stalling due to full memory queues. These stalls occur when warps are ready to issue memory operations (to global, local, or shared memory via the Load Store Unit), but are temporarily prevented from doing so because the dedicated instruction queues leading to these execution units are full, often due to high demand or downstream backpressure in the memory system. At selectivity 1.0, stalls due to full memory instruction queues stalls nearly double to 10.30% with Cascaded compared to 5.64% when no compression is used. At low selectivity (0.02), however, these specific queue stalls are minimal for Cascaded at 0.23% (even slightly lower than the 0.33% without compression). Cascaded generates substantial memory traffic because the Cascaded encoder, as implemented in nvCOMP, launches up to four separate kernels per data block: the main decompression kernel and three additional ones depending on intermediate data types. Each of these kernels writes intermediate results back to shared memory. These stages are necessary for decoding nested encodings like RLE and delta encodings. While

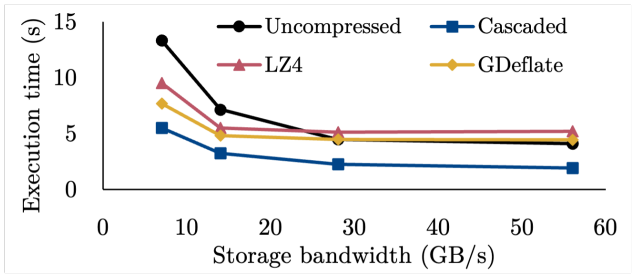


(a) Execution time of SSB queries with data stored on 1 NVMe drive.

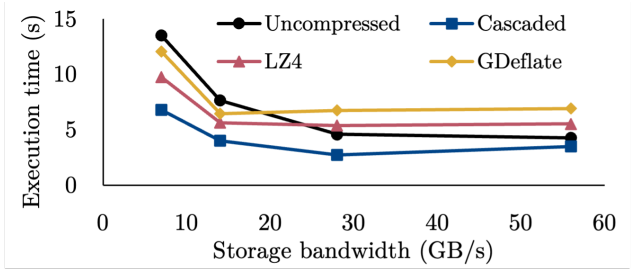


(b) Execution time of SSB queries with data stored on 8 NVMe drives

Figure 3: Execution time of SSB queries with data stored in different number of NVMe drives, normalized by execution time without compression.



(a)



(b)

Figure 4: Execution time of SSB Q1.1 (a) and Q3.1 (b) with varying storage bandwidth using different compression schemes.

prior work [42] has already noted this overhead, our results show that it still improves out-of-memory query execution times across most queries at most selectivities in our microbenchmark. Overall, Cascaded is generally effective, but its multi-stage decompression process inherently creates significant memory traffic, which can interfere with the query processing kernel's own memory demands; further kernel fusion for Cascaded could mitigate this interference and enhance performance.

With LZ4 decompression, more stalls are due to compute, which includes waiting for compute resources to become available or fixed-latency execution dependencies: the percentage of cycles where warps are stalled waiting for compute resources is substantially elevated with LZ4, reaching 40.52% at selectivity 0.02 and 27.87% at selectivity 1.0, far exceeding the 29.6% at selectivity 0.02 and 10.01% at selectivity 1.0 with Cascaded and 10.15% and 3.53% observed without compression. This highlights the computationally intensive nature of the LZ4 algorithm with respect to Cascaded. This implies that LZ4's primary mode of interference with query processing kernels is likely through competition for the SMs arithmetic resources rather than by causing significant congestion in memory access queues.

4.2 Star Schema Benchmark

In this section, we identify the impact of lightweight and heavyweight compression on the full SSB benchmark. In particular, we show how the performance varies depending on the storage bandwidth, as greater storage bandwidth consequently requires greater decompression throughput.

Figure 3a plots the execution time for all SSB queries normalized to the execution time without using compression when the data is stored on a single NVMe drive. In this scenario, the bandwidth of the NVMe is the primary bottleneck in query execution. As a result, the normalized execution time remains stable across each query group (aligning to the compression ratio of each algorithm on the scanned columns) since the storage bandwidth constraint dominates performance, masking the effects of query processing intensity. Cascaded consistently outperforms both LZ4 and GDeflate, due to its significantly higher compression ratio.

Figure 3b, on the other hand, demonstrates a different behavior. With increased storage bandwidth, the system transitions away from being purely storage-bound, and query complexity begins to play a more significant role in execution time. This trend aligns with what we observed in the microbenchmark experiments (Figure 2): as storage bandwidth increases, the computational cost of decompression begins to compete with query execution for resources, resulting in performance degradation. For example, Q3.1 is slower than Q3.2, Q3.3, and Q3.4, despite scanning the same volume of data with the same compression ratio. This is because the joins of Q3.1 are less selective than the other queries.

Figure 3a and fig. 3b show that, unlike the Q*J queries, GDeflate demonstrates a substantial speedup on Q1.1, due to a 1.96 compression ratio compared to LZ4s 1.46 for this query (see Table 1). This is primarily due to GDeflate's ability to more effectively compress `lo_discount`, which contains only 11 unique values. The entropy coding in GDeflate efficiently encodes these repeated values, achieving a higher data reduction than LZ4.

To further illustrate the impact of compression on query execution, fig. 4 presents the execution time for Q1.1 and Q3.1 across varying storage bandwidths using different compression schemes.

For Q1.1 (fig. 4a), compression provides a significant performance boost at lower bandwidth, while heavyweight decompression results in slower query execution time than no compression at all for greater bandwidths. In contrast, Cascaded maintains its speed up even for higher bandwidths.

Q3.1 (fig. 4b) exhibits a similar trend, but the query performance using heavyweight compression begins degrading at lower bandwidths than Q1.1. Further, even Cascaded begins to degrade at the highest bandwidth. Relative to Q1.1, Q3.1 is a processing-intensive query that includes three joins and a groupby, and so is more susceptible to interference from decompression.

4.3 Roofline Analysis

The roofline model [45] relates computational performance to memory bandwidth constraints, helping to identify whether a workload is compute-bound or memory-bound based on its operational intensity. In this work, we use a roofline to model storage-boundedness instead by measuring the volume of data loaded from storage and the instructions executed per byte of data loaded from storage. This allows us to characterize the resource bottleneck in query processing when the data is streamed from the NVMe drives to the GPU during query execution. Naraparaju et al. [30] introduce a similar compression-enabled roofline analysis to demonstrate how compression can shift a CPU-based matrix multiplication from being memory-bound towards compute-bound. While our study focuses on GPU query processing systems that are storage-bound because of the storage/interconnect bandwidth rather than memory constraints, both approaches highlight compression's potential to mitigate data movement bottlenecks to achieve more processing-bound performance.

GPU profiling strategy. Similar to Cao et al. [12], we use Nsight Compute [5] to obtain kernel execution metrics. However, unlike Cao et al., where each kernel is launched once when the entire dataset is already in GPU-memory, our system invokes decompression and query execution kernels for each row group of data moved to the GPU. To form a complete view of query execution, we aggregate performance metrics across kernel invocations within a query. For each query, we aggregate profiling data across more than 70K and 310K, in the uncompressed and compressed cases, respectively, kernel invocations of all types.

Rooflines. Figure 5 shows the roofline model characterization for the execution of SSB queries with and without cascaded decompression. Using four drives (23.76 GB/s of bandwidth when measured with Nvidia's gdsio utility [4]), uncompressed execution has an operational intensity ranging from 1.4-3.7. With compression, this increases to an operational intensity of 67.1-105.2. Execution with decompression results in 9.4 to 28.6 times more executed instructions. In comparison, the speedup for these queries by using compression ranges from a 1.4× speedup to a 2.0× speedup. The operational intensity of Q1.1 increases the most, as it is the least processing-intensive without compression, and also experiences the largest speedup in execution time.

More precisely, Q1.1 exhibits the lowest throughput in the uncompressed setting because it is the least processing-intensive query. Its limited compute demand (only one join) leads to lower GPU utilization. Consequently, performance is tightly bound by storage bandwidth. In the compressed case, most of the workload shifts to decompression (as we said, we notice from 9.4× to 28.6× more executed instructions). This makes the operational profiles of all queries more similar, as the same decompression step dominates execution. However, Q1.1 stands out in the compressed setting by achieving the highest throughput. This is because Cascaded achieves the highest compression ratio on the columns used in Q1.1 (as shown in Table 1), meaning decompression and query processing operate on less (compressed) data compared to the other queries. As a result, executing query Q1.1, storing the dataset compressed with Cascaded, appears further to the top-right of the roofline plot, combining higher operational intensity with greater throughput, overcoming the storage bandwidth limitations.

This demonstrates that uncompressed execution heavily underutilizes the GPU hardware. Thus despite the high instruction count overhead of decompression, the reduced volume of data transferred both increases hardware utilization and alleviates the data movement bottleneck, ultimately improving query performance.

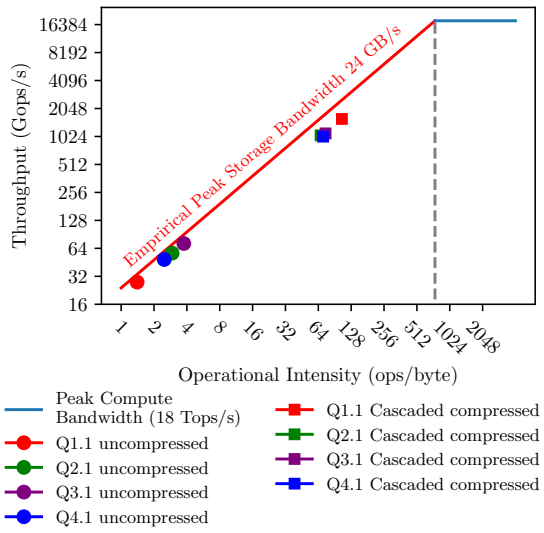


Figure 5: Roofline plot of four SSB queries with and without compression using 4 NVMe drives. The vertical dashed line marks the boundary between storage-bound (left) and compute-bound execution (right).

5 Related Work

Our work focuses on streaming compressed data to a single GPU. Caching mechanisms [34, 35], CPU-GPU co-execution [47], pruning [10], and multi-GPU scenarios [46] remain important avenues for future research.

GPU DBMS and data compression. It is well known that in processing larger-than-GPU datasets, the dominant bottleneck is transferring data to the GPU rather than executing the relational operators themselves [20]. Recent work [38] demonstrates that this challenge remains critical, especially when designing efficient data movement and decompression pipelines.

Li et al. [26] first demonstrated the potential of data compression to address the bandwidth discrepancy between GPUs and slower storage devices in big data analytics. They introduce HippogriffDB, a GPU-accelerated OLAP system that stores data in a compressed format and leverages GPU decompression to trade computation for greater effective I/O bandwidth. It employs run length encoding [43], dictionary encoding, Huffman encoding [29], and delta encoding to compress tables. Compressing data as much as possible is not always convenient because it affects query time (more time is needed for decompression).

Shanbhag et al. [42] build on top of the tile-based execution model introduced in Crystal [41], combining it with lightweight compression techniques. Tiles of data are loaded into shared memory, enabling direct access for subsequent accesses within the same kernel, which minimizes redundant memory accesses. They integrate lightweight compression methods, such as GPU-FOR, GPU-DFOR, and GPU-RFOR, the tile-based decompression, into the Crystal framework. By leveraging GPU-shared memory and implementing cascaded compression schemes within a single kernel, the approach outperforms traditional methods that rely on separate kernel invocations for each decompression layer. This design also incorporates the planner from [18] to select optimal compression schemes based on column properties, further enhancing query performance.

We focus on nvComp Cascaded over GPU-* [42] because it achieves comparable compression ratios, making it a fair baseline for evaluating compression impact. While GPU-* offers faster decompression through in-GPU-memory microarchitectural optimizations, our focus is on a broader system-level perspective in which data is streamed from storage and storage bottlenecks dominate performance.

A recent paper [12] conducts a detailed analysis of the performance bottlenecks in the original Crystal without compression [41] and other GPU-DBMS on uncompressed data, leveraging microarchitectural metrics and the roofline model to identify areas for improvement. The analysis reveals inefficiencies in memory bandwidth utilization (particularly DRAM or L2 cache bandwidth). They demonstrate opportunities for optimization related to kernel fusion, thread termination, and cache utilization through an improved version of Crystal (Crystal-opt).

Afroozeh et al. [7] focuses on reducing memory pressure and improving decompression efficiency by integrating a new compression scheme called FastLanesGPU (FLS-GPU). The main idea is to have tiles of data distributed in a round-robin manner over subsequent lanes. Thus, a data-parallel kernel can produce subsequent values from subsequent lanes without needing (expensive) inter-lane data transfers. FLS-GPU currently only implements bit-packing. Other common encodings (specifically DELTA, RLE, and DICT) have not yet been ported to CUDA.

GOLAP [10] proposes a GPU-in-data-path architecture to accelerate table scans by performing data pruning on the GPU and streaming data compressed with heavyweight compression directly from SSDs to the GPU, where it is decompressed on the fly as part of a table scan. GOLAPs GPU-optimized pruning techniques use computationally intensive summaries (e.g., histograms) to reduce the volume of data loaded from storage. Our study further reinforces the importance of compression schemes tailored for GPUs, expanding the design space to highlight the cost of decompression and competing demands for hardware resources.

6 Conclusion and future work

This paper analyzes the performance of pipelined decompression GPU query processing over NVMe-resident data sets. With limited storage bandwidth, we find that the execution time of SSB queries over compressed data improves over no compression in line with compression ratios for both lightweight and heavyweight compression. However, for storage bandwidth near the interconnect bandwidth limit, lightweight compression consistently outperforms heavy-weight compression, and heavy-weight decompression can even perform worse than no compression.

Through microbenchmarks that vary decisive query parameters such as selectivity and number of joins, controlling the average amount of work necessary to process each tuple, we identify when decompression performance degrades and becomes worse than no compression. Finally, our storage roofline analysis indicates that query execution with uncompressed data heavily underutilizes the GPU. Thus, while lightweight compression dramatically increases the executed instruction count, the reduced volume of data moved results in improved query execution times for SSB queries.

One promising direction is the development of even more compressed, compute-intensive, GPU-optimized compression techniques specifically tailored for columnar data. As shown in our roofline analysis (Figure 5), current workloads, even with decompression, remain on the storage-bound side of the spectrum. This suggests the possibility of compressing the data even more, and shifting further right on the roofline, i.e., to increase operational intensity by introducing additional lightweight transformations or compression layers, without hitting compute limitations. Future systems could exploit this to improve overall throughput.

Moreover, future work should extend the scope of query patterns analyzed. While this paper focused on scan- and join-heavy analytical workloads, exploring other sources of query complexity (such as aggregation, grouping, and window functions) could provide further insights into the interaction between decompression and query processing. These patterns may introduce different contention dynamics on the GPU.

The methodologies employed in this paper will be increasingly needed to understand and quantify the effectiveness of compression with future hardware, helping to navigate data movement bottlenecks. With higher bandwidth interconnects, such as new PCIe versions or NVLink, it is possible to attain greater bandwidth between storage and the GPU, shifting the performance inflection point (relative to query complexity). Consequently, less processing-intensive queries could perform worse due to GPU resource contention or hitting the limit of GPU decompression. Further, while next-generation architectures like NVIDIA's Blackwell include dedicated decompression engines designed to accelerate modern formats, which will likely shift the performance bottleneck in the other direction, benefiting more complex queries. For different ratios of GPU decompression throughput to interconnect bandwidth, the performance inflection point of using compression shifts. As interconnect and GPU technologies evolve, this ratio will vary, requiring re-evaluation of the benefit of compression.

References

[1] 2022. https://docs.nvidia.com/cuda/cub/index.html
[2] 2024. https://docs.nvidia.com/cuda/nvcomp/
[3] 2024. AMD EPYC™ 9965. https://www.amd.com/en/products/processors/server/epyc/9005-series/amd-epyc-9965.html
[4] 2024. GPUDirect Storage Benchmarking Tools. https://docs.nvidia.com/gpudirect-storage/configuration-guide/index.html#gpudirect-storage-benchmarking-tools
[5] 2025. https://developer.nvidia.com/nsight-compute
[6] 2025. GPUDirect Storage. https://docs.nvidia.com/gpudirect-storage/
[7] 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* (Santiago, Chile) *(DaMoN '24)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3662010.3663450
[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* (Pittsburgh, Pennsylvania, USA) *(GPGPU-3)*. Association for Computing Machinery, New York, NY, USA, 94–103. https://doi.org/10.1145/1735688.1735706
[9] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 283–296. https://doi.org/10.1145/1559845.1559877
[10] Nils Boeschen, Tobias Ziegler, and Carsten Binnig. 2024. GOLAP: A GPU-in-Data-Path Architecture for High-Speed OLAP. *Proc. ACM Manag. Data* 2, 6, Article 237 (Dec. 2024), 26 pages. https://doi.org/10.1145/3698812
[11] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proc. VLDB Endow.* 13, 12 (July 2020), 2649–2661. https://doi.org/10.14778/3407790.3407851
[12] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proc. VLDB Endow.* 17, 3 (Nov. 2023), 441–454. https://doi.org/10.14778/3632093.3632107
[13] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2016. Better bitmap performance with Roaring bitmaps. *Softw. Pract. Exper.* 46, 5 (May 2016), 709–719. https://doi.org/10.1002/spe.2325
[14] Periklis Chrysogelos. 2022. *Efficient analytical query processing on CPU-GPU hardware platforms*. phd. EPFL, Lausanne. https://infoscience.epfl.ch/handle/20.500.14299/190430
[15] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.* 12, 5 (2019), 544–556. https://doi.org/10.14778/3303753.3303760
[16] P. Deutsch. 1996. RFC1951: DEFLATE Compressed Data Format Specification version 1.3.
[17] EPFL DIAS. 2025. Proteus: A Database Engine for Heterogeneous Environments. https://github.com/epfl-dias/proteus. Accessed: 2025-02-01.
[18] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database compression on graphics processors. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 670–680. https://doi.org/10.14778/1920841.1920927
[19] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1603–1618. https://doi.org/10.1145/3183713.3183734
[20] Emily Furst, Mark Oskin, and Bill Howe. 2017. Profiling a GPU database implementation: a holistic view of GPU resource utilization on TPC-H queries. In *Proceedings of the 13th International Workshop on Data Management on New Hardware* (Chicago, Illinois) *(DAMON '17)*. Association for Computing Machinery, New York, NY, USA, Article 3, 6 pages. https://doi.org/10.1145/3076113.3076119
[21] Andrea Guerra, Giorgio Vinciguerra, Antonio Boffa, and Paolo Ferragina. 2025. Learned Compression of Nonlinear Time Series with Random Access . In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1579–1592. https://doi.org/10.1109/ICDE65448.2025.00122
[22] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. 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/p16-haas-cidr20.pdf
[23] Maximilian Kuschewski, Jana Giceva, Thomas Neumann, and Viktor Leis. 2024. High-Performance Query Processing with NVMe Arrays: Spilling without Killing Performance. *Proc. ACM Manag. Data* 2, 6 (2024), 238:1–238:27. https://doi.org/10.1145/3698813
[24] Viktor Leis. 2024. LeanStore: A High-Performance Storage Engine for NVMe SSDs. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 4536–4545. https://doi.org/10.14778/3685800.3685915
[25] A. Lempel and J. Ziv. 1976. On the Complexity of Finite Sequences. *IEEE Transactions on Information Theory* 22, 1 (1976), 75–81. https://doi.org/10.1109/TIT.1976.1055501
[26] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: balancing I/O and GPU bandwidth in big data analytics. *Proc. VLDB Endow.* 9, 14 (Oct. 2016), 1647–1658. https://doi.org/10.14778/3007328.3007331
[27] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2024. LeCo: Lightweight Compression via Learning Serial Correlations. *Proc. ACM Manag. Data* 2, 1, Article 65 (March 2024), 28 pages. https://doi.org/10.1145/3639320
[28] Sina Meraji, Berni Schiefer, Lan Pham, Lee Chu, Peter Kokosielis, Adam Storm, Wayne Young, Chang Ge, Geoffrey Ng, and Kajan Kanagaratnam. 2016. Towards a Hybrid Design for Fast Query Processing in DB2 with BLU Acceleration Using Graphical Processing Units: A Technology Demonstration. 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, 1951–1960. https://doi.org/10.1145/2882903.2903735
[29] Alistair Moffat. 2019. Huffman Coding. *ACM Comput. Surv.* 52, 4, Article 85 (Aug. 2019), 35 pages. https://doi.org/10.1145/3342555
[30] Ramasoumya Naraparaju, Tianyu Zhao, Yanting Hu, Dongfang Zhao, Luanzheng Guo, and Nathan Tallent. 2025. Shifting Between Compute and Memory Bounds: A Compression-Enabled Roofline Model. In *Proceedings of the SC '24 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis* (Atlanta, GA, USA) *(SC-W '24)*. IEEE Press, 309–316. https://doi.org/10.1109/SCW63240.2024.00047
[31] Gonzalo Navarro. 2022. Indexing Highly Repetitive String Collections, Part I: Repetitiveness Measures. *ACM Comput. Surv.* 54, 2 (2022), 29:1–29:31. https://doi.org/10.1145/3434399
[32] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. 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/p29-neumann-cidr20.pdf
[33] Hamish Nicholson, Periklis Chrysogelos, and Anastasia Ailamaki. 2022. HPCache: Memory-Efficient OLAP Through Proportional Caching. In *International Conference on Management of Data, DaMoN 2022, Philadelphia, PA, USA, 13 June 2022*, Spyros Blanas and Norman May (Eds.). ACM, New York, NY, USA, 7:1–7:9. https://doi.org/10.1145/3533737.3535100
[34] Hamish Nicholson, Periklis Chrysogelos, and Anastasia Ailamaki. 2024. HPCache: memory-efficient OLAP through proportional caching revisited. *VLDB J.* 33, 6 (2024), 1775–1791. https://doi.org/10.1007/S00778-023-00828-7
[35] Hamish Nicholson, Aunn Raza, Periklis Chrysogelos, and Anastasia Ailamaki. 2023. HetCache: Synergising NVMe Storage and GPU acceleration for Memory-Efficient Analytics. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p84-nicholson.pdf
[36] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5895)*, Raghunath Othayoth Nambiar and Meikel Poess (Eds.). Springer, 237–252. https://doi.org/10.1007/978-3-642-10424-4_17
[37] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: a fast, scalable, in-memory time series database. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1816–1827. https://doi.org/10.14778/2824032.2824078
[38] Cristian Peñaranda, Carlos Reaño, and Federico Silla. 2024. Hybrid-Smash: A Heterogeneous CPU-GPU Compression Library. *IEEE Access* 12 (2024), 32706–32723. https://doi.org/10.1109/ACCESS.2024.3371253
[39] Aunn Raza, Periklis Chrysogelos, Panagiotis Sioulas, Vladimir Indjic, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. 2020. GPU-accelerated data management under the test of time. 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/p18-raza-cidr20.pdf
[40] Mark A. Roth and Scott J. Van Horn. 1993. Database compression. *SIGMOD Rec.* 22, 3 (Sept. 1993), 31–39. https://doi.org/10.1145/163090.163096
[41] 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 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595
[42] Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-Based Lightweight Integer Compression in GPU. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1390–1403. https://doi.org/10.1145/3514221.3526132

[43] Jukka Teuhola. 1978. A compression method for clustered bit-vectors. *Inform. Process. Lett.* 7, 6 (1978), 308–311. https://doi.org/10.1016/0020-0190(78)90024-8

[44] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (2024), 3694–3706. https://www.vldb.org/pvldb/vol17/p3694-saxena.pdf

[45] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[46] Bobbi Yogatama, Weiwei Gong, and Xiangyao Yu. 2025. Scaling your Hybrid CPU-GPU DBMS to Multiple GPUs. *Proc. VLDB Endow.* 17, 13 (Feb. 2025), 4709–4722. https://doi.org/10.14778/3704965.3704977

[47] Bobbi W. Yogatama, Weiwei Gong, and Xiangyao Yu. 2022. Orchestrating Data Placement and Query Execution in Heterogeneous CPU-GPU DBMS. *Proc. VLDB Endow.* 15, 11 (2022), 2491–2503. https://doi.org/10.14778/3551793.3551809

[48] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *Proc. VLDB Endow.* 6, 10 (Aug. 2013), 817–828. https://doi.org/10.14778/2536206.2536210

[49] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proc. VLDB Endow.* 17, 2 (2023), 148–161. https://www.vldb.org/pvldb/vol17/p148-zeng.pdf

Received 21 March 2025