Do GPUs Really Need New Tabular File Formats?

Jigao Luo*
TU Darmstadt
Darmstadt, Germany

Qi Chen
TU Darmstadt
Darmstadt, Germany

Carsten Binnig
TU Darmstadt & DFKI & hessian.AI
Darmstadt, Germany

Abstract

Apache Parquet is the de facto columnar file format in modern analytical systems, yet its configuration guidelines remain largely shaped by CPU-centric execution engines. As GPU-accelerated data processing becomes increasingly prevalent, Parquet files optimized by default for CPU-oriented engines often severely underutilize GPU hardware, unnecessarily turning GPU scans into an artificial performance bottleneck.

In this work, we systematically study how Parquet configurations affect GPU scan performance, and show that poor Parquet performance on GPUs is mostly a consequence of suboptimal configuration choices. By applying GPU-aware configurations, we achieve up to 125 GB/s effective read bandwidth using fully compliant Parquet files, without switching to a new file format or sacrificing the interoperability benefits of the de facto standard.

CCS Concepts

• **Information systems** → *Online analytical processing engines.*

Keywords

File Format, GPU, NVMe SSDs, Encodings, Compression, Parquet

ACM Reference Format:
Jigao Luo, Qi Chen, and Carsten Binnig. 2026. Do GPUs Really Need New Tabular File Formats?. In *22nd International Workshop on Data Management on New Hardware (DaMoN '26), May 31–June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3789237.3809125

1 Introduction

Parquet Dominance. Over the past decade, Parquet has become the de facto columnar file format in modern analytics systems. It is widely adopted across analytical databases and query engines such as DuckDB [34], Velox [32], and DataFusion [20], all of which support direct querying of Parquet files. The format exposes several configuration parameters that affect performance, including page counts, row group sizes, encoding, and compression. Yet the default parameter values were chosen based on CPU-oriented access patterns and I/O behavior [9, 10, 23, 45]. These defaults are used for most Parquet files, even when those files will be used for GPU scans.

GPU Bottleneck With Parquet. Despite extensive CPU-side testing, Parquet's reported performance on GPUs remains subpar. In practice, Parquet reading is a major bottleneck in GPU-accelerated analytics: the NVIDIA RAPIDS team reports that about **85%(!)** of TPC-H benchmark runtime is spent on Parquet scan rather than on other query operators [17]. This naturally raises our central research question: *do GPUs really need new tabular file formats?*

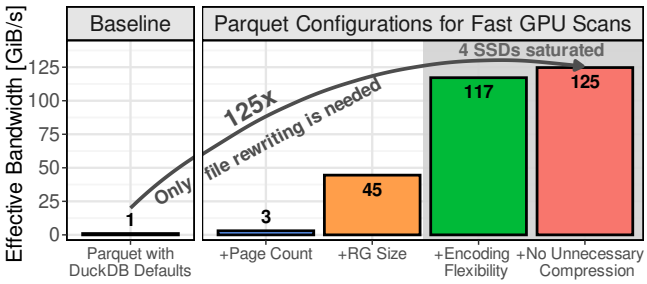

Figure 1: GPU Parquet scan on TPC-H SF300 `lineitem` with 4 SSDs: file configuration impact on effective read bandwidth.

Why Is Parquet Not Optimal for GPUs? As shown in Figure 1 (left), Parquet written with the default configuration delivers poor GPU scan performance, while GPU-aware configurations yield dramatic improvements. Simply by rewriting Parquet files with GPU-friendly settings, the effective bandwidth increases up to 125 GB/s. As such, we believe that in many cases, Parquet itself is fully capable of achieving high GPU performance without requiring a new file format for GPUs.

Need for New File Formats? The goal of this work is to push the existing Parquet format to its practical limits before considering entirely new file-format designs for GPUs. Such formats would need to outperform the optimized Parquet configuration shown in Figure 1, rather than merely improving over the defaults, a direction beyond the scope of this work. Our aim instead is to thoroughly explore the performance potential already available within the Parquet specification and to establish a strong baseline for future GPU file formats. Recent work has also proposed new file formats for GPUs, but these efforts mainly target in-GPU-memory decoding [1, 2, 13, 39, 40]. Their approach differs from this work in two ways. First, we evaluate practical scenarios by reading directly from SSDs rather than GPU memory. Second, we preserve the existing Parquet ecosystem while maintaining compatibility and interoperability.

Contributions. In this short paper, we present a set of insights and the first systematic evaluation of how Parquet configuration choices affect GPU scan performance. We also provide a rewriter tool[1] that transforms Parquet files into arbitrary configurations. Collectively, our insights and tool provide the first practical guidance on optimizing Parquet for GPU databases, enabling substantial acceleration without requiring a completely new file format.

*Correspondence goes to jigao.luo@tu-darmstadt.de.

[1]The Parquet rewriter tool is available at: https://github.com/DataManagementLab/ParquetRewriter.

This work is licensed under a Creative Commons Attribution 4.0 International License.
DaMoN '26, Bengaluru, India
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2455-8/26/05
https://doi.org/10.1145/3789237.3809125

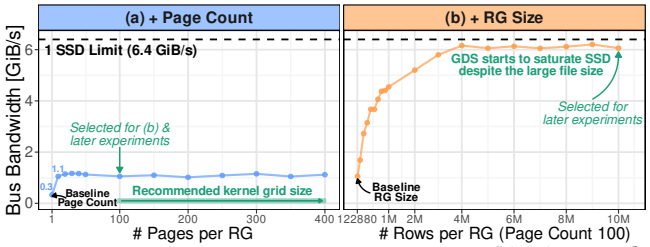

Figure 2: GPU Parquet scan on TPC-H SF300 `lineitem` with one SSD: storage bus bandwidth of different file configurations. Left: varying page counts. Right: varying rows per RG.

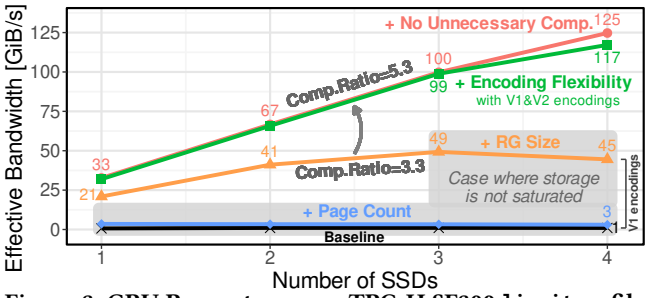

Figure 3: GPU Parquet scan on TPC-H SF300 `lineitem`: file configuration and SSD scaling effects on effective bandwidth.

2 Parquet in GPU Databases

Apache Parquet [38] is a widely used columnar file format for analytical processing. A row group (RG) contains one column chunk per column, and each column chunk is divided into pages. Pages may use different encodings from both the older V1 and newer V2 schemes. After encoding, compression is applied at the column chunk level using algorithms such as Snappy, Zstandard, or GZIP.

NVIDIA RAPIDS cuDF [35] is the only GPU-accelerated library that reads Parquet directly on the GPU, with both decompression and decoding executed as GPU kernels. With GPUDirect Storage (GDS) [11], cuDF reads Parquet directly from storage into GPU memory. As a foundational library, cuDF underpins several GPU-accelerated analytical systems, including Theseus [7], Sirius [43], Velox-cuDF [17], and PystachIO [24]. Among them, PystachIO is the only system that fully overlaps I/O and GPU computation. In this work, we use PystachIO to investigate Parquet configuration and optimizations on GPUs.

3 Pushing Limits of Parquet for GPUs

In this section, we present four key insights into Parquet configuration and its effects on GPU scans. Our setup uses one NVIDIA A100 GPU reading the largest TPC-H SF300 table, `lineitem`, from local NVMe SSDs via GDS. Guided by these insights, we develop a rewriter tool for simple and flexible reconfiguration of Parquet files.

Baseline. The baseline Parquet file, used as the initial input to our rewriter, is generated by DuckDB using its default settings. Under these defaults, each column chunk contains a single page, each RG contains 122880 rows [9, 10] with only V1 encodings. The page count and RG size follow DuckDB's implementation best practice of mapping RGs, not pages, to CPU cores for parallelism. DuckDB defaults to older Parquet V1 encodings to maintain broad compatibility with systems that may not support reading newer Parquet V2 encodings.

Increase Page Count. To read Parquet files on GPUs, we use cuDF, which maps the number of pages to the grid size in its GPU kernel launch parameters. In effect, cuDF parallelizes across both RGs and pages to exploit the GPU's massive parallelism. This leads to an important consequence: if a Parquet file contains too few pages, only a small portion of the GPU becomes active during decoding. To achieve high GPU utilization, the page count specified at file generation should match the kernel grid size recommended for modern GPUs – typically well above 100.

As shown in Figure 2(a), increasing the page count from 1 raises the read bandwidth. The gain comes from the decoding kernel running more efficiently. However, increasing the page count further does not improve storage bandwidth, a limitation we examine next. In the following experiments, we use a page count of 100 as a reasonable choice for kernel grid size. There is no single universally optimal page count; the key is simply to avoid small values.

> **Insight 1:** Given the characteristics of GPU decoding kernels, increasing the page count to 100 or above is recommended.

Improve RG Size. The GPU I/O stack in GDS behaves quite differently from the CPU I/O stack, leading to a different choice of optimal RG size. DuckDB determines its RG size through tailored microbenchmarking [9, 10], producing column chunks on the order of 100 KB once compressed and encoded. For GDS reads, however, such small I/O units are suboptimal, as MiB-scale transfers tend to be more efficient [8, 36]. This motivates using a much larger RG size so that each column chunk reaches an I/O size that allows GDS to saturate storage.

The small I/O size implied by DuckDB's RG size explains why Figure 2(a) stops improving; as shown in Figure 2(b), increasing the RG size beyond 4M rows helps GDS saturate one SSD. In the following experiments, we use a larger RG size of 10M rows, as the SSD must remain saturated even when our subsequent techniques further reduce the I/O size. This also reinforces our Insight 1: such a large RG naturally requires fine-grained page-level parallelism.

> **Insight 2:** Given the GPU I/O stack, million-row RG sizes are preferred to allow GDS to saturate storage.

Encoding Flexibility. Most Parquet writers apply a single V1 encoding to an entire column, unlike proprietary formats that use per-chunk encoding strategies tailored to local data distributions [8, 18, 21]. Extending these existing ideas to our Parquet rewriter, we allow each column chunk to explore a wider range of candidate encodings in Parquet V1 and V2, while finalizing only the one that yields the smallest encoded size. The rationale for choosing the smallest size is to alleviate the I/O bottleneck [8, 14, 16, 24, 28, 42].

Our current encoding-selection strategy explores all Parquet valid encodings for a specific data type, rather than relying on the sampling-based approaches used in prior work [8, 18]. This is still practically feasible because the search space is small, with fewer than five candidate Parquet encodings for any given data type. Moreover, the overhead of the current encoding selection is relatively low. We provide further discussions of this overhead and

GPU encodings in Section 5, while leaving sampling- and heuristic-based encoding selections to our future work.

As shown in Figure 3, enabling encoding flexibility improves the compression ratio, reflecting the size reduction after encoding and compression. Note that in Figure 3, we report *effective bandwidth*, computed as the logical raw data size after decoding and decompression divided by the scan runtime. We switch to this metric because improving RG size alone already saturates storage bandwidth (see Figure 2(b)). This metric thus captures the rate at which logical raw data is scanned into GPU memory. With encoding flexibility, effective bandwidth scales more linearly with the number of SSDs, especially when more than two SSDs are used, compared with the case of the RG size optimization shown in Figure 3.

The increase in effective bandwidth from our encoding flexibility is mainly due to two reasons. First, the smaller encoded size achieved by better encodings efficiently mitigates the I/O bottleneck, since less data needs to be read from storage. Prior Parquet V1 encodings are typically limited to plain or dictionary encoding, which do not fully exploit the underlying data distributions, whereas Parquet V2 encodings, such as delta encoding, often can. The resulting improvement in compression ratio is also annotated in Figure 3. Second, Parquet V2 encodings are also efficient on GPUs in both decoding bandwidth and resource usage, thereby addressing the common criticism of Parquet encodings [2, 18, 19, 45]. This is also consistent with feedback from the cuDF team, who noted that delta encoding `DELTA_BINARY_PACKED` in V2 achieves high decoding bandwidth for integer data types.

> **Insight 3:** Flexibly choosing modern Parquet encodings to achieve higher size reduction improves GPU scan performance.

No Unnecessary Compression. A practical limitation of compression arises when it is applied blindly, even in cases where it yields no size reduction. We believe this contributes to the common criticism of Parquet compressions [18, 19, 45, 46], where decompression overhead slows down query execution. In practice, compression is a tradeoff between reducing I/O size in storage and adding decompression cost during reading, so it should be applied selectively rather than unconditionally. Following this principle, we finalize compression for a column chunk only when its size reduction exceeds a chosen threshold (10% in our experiments); otherwise, the column chunk remains uncompressed. We choose this 10% threshold empirically: it helps skipping cases where there is no gain, or where the gain is too small to justify the additional decompression cost. We acknowledge that an optimal threshold should ideally be determined by considering the tradeoff between decompression bandwidth and I/O bandwidth, and we leave this as future work.

As shown in Figure 3, avoiding unnecessary compression improves performance only when the GPU reads Parquet from four SSDs. This is expected: reading four SSDs, the GPU tends to become compute-bound as many decompression and decoding kernels are queued. Removing unnecessary decompression allows other useful kernels to run earlier. The effect disappears with fewer SSDs because the workload becomes I/O-bound, and the GPU has ample compute capacity for decompression. Nicholson et al. observed similar behavior, where decompression degraded performance once storage bandwidth approached the PCIe limit [28]. Although the gains are modest, we still recommend avoiding unnecessary compression, as the file must be read efficiently across many possible cases.

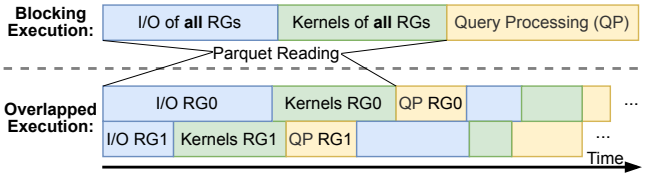

Figure 4: High-level comparison of blocking and overlapped execution. (1) The Parquet reading stage consists of storage I/O and GPU kernels. (2) Full query execution comprises Parquet reading followed by query processing operators.

> **Insight 4:** Skip unnecessary compression if no size reduction.

4 Overlapping I/O and GPU Computation

Given the optimized file configuration, a remaining question in the data path is how file-level improvements translate into gains at the query level. To this end, we next discuss overlapping Parquet reading with GPU query processing. We focus on overlap because it is a key technique for mitigating I/O bottlenecks when GPUs process datasets that exceed memory capacity and are therefore transferred from storage, and it is widely used in recent GPU systems [8, 24, 42, 44].

In this section, we continue to use a single NVIDIA A100 GPU with one SSD and evaluate the query runtime of our GPU query engine, PystachIO. To this end, we study two I/O-intensive TPC-H queries: Query 6, a single-table aggregation query that essentially scans the full Parquet file, and Query 12, which includes a join between the two largest tables in TPC-H.

4.1 Overlapped Parquet Reading

We additionally study GPU Parquet reader optimizations, with a more detailed analysis given in [24]. The key finding is that an overlapped reader is a prerequisite for fully realizing both the benefits of the optimized file configuration and the GPU performance potential. It also serves as a necessary foundation for the performance results reported in this work. Figure 4 illustrates this through a comparison of blocking and overlapped Parquet reading. In the blocking design, decompression and decoding kernels can be launched only after the entire I/O phase completes, leaving the GPU entirely idle during that time. In contrast, overlapping I/O with GPU computation not only enables kernel execution during the I/O phase but also helps avoid out-of-memory errors by processing data at RG granularity.

This work on file configuration and the study of reader design in [24] provide complementary perspectives, and both are necessary for understanding GPU Parquet scan performance. We therefore view this performance as determined by two tightly coupled dimensions: **file configuration** and **reader design**. Both are necessary, and neither alone is sufficient: even with our optimized overlapped reader, performance remains poor if the file configuration itself is unfavorable to GPUs, as illustrated in Figures 3 and 5. In Figure 5, the baseline and increased page count both lead to very slow runtime, whereas increasing RG size brings performance much closer to the optimized file configuration, which incorporates the encoding and compression optimizations introduced earlier.

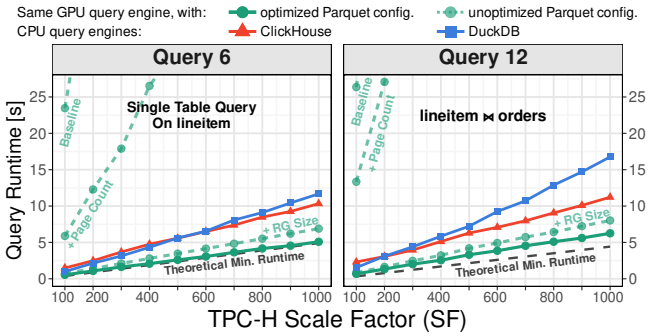

Figure 5: Dataset scalability up to TPC-H SF1000 on one SSD: query performance of the GPU query engine PystachIO. (1) Comparison across different Parquet configurations, with extremely high runtimes omitted. The optimized case combines `+Encoding Flexibility` with `+No Unnecessary Compression`. (2) Comparison against CPU systems, ClickHouse and DuckDB, on the same optimized Parquet files.

4.2 Overlapped Query Processing

With both the file configuration and the reader optimized, we further study the effect of overlap on end-to-end GPU query processing. A more detailed analysis is given in [24]. The key idea is to extend overlap beyond Parquet reading to additional query operators, increasing the scope of overlap between I/O and computation. Once I/O dominates execution time, broadening the overlap helps improve query performance by further hiding I/O latency. With overlapped query processing in Figure 4, each RG produced by Parquet reading is directly consumed by a query operator, e.g., on the build side of a hash join.

To demonstrate the effectiveness of overlapped GPU query processing, we compare our GPU system against two CPU systems, shown as solid lines in Figure 5. All three systems use the same Parquet files but different hardware setups: the CPU systems run on an AMD EPYC 9654P 96-Core Processor without a GPU. To indicate proximity to the theoretical lower bound of query runtime, the gray reference line is computed as the total read size from storage divided by SSD bandwidth. Overall, the GPU system outperforms the CPU systems by a large margin and remains close to the theoretical minimum, demonstrating that our file-level optimizations are effective and already translate into substantial improvements at the query level. Since CPU system performance does not vary substantially across different file configurations, we use the same optimized configuration for a fair comparison. Our GPU engine remains close to the theoretical minimum for both queries.

5 Discussions

Overheads of Parquet Rewriting. By using Rust and an efficient multithreaded implementation, our rewriter fully utilizes modern CPUs. Rewriting typically completes within minutes for a 100 GB dataset. For many systems, an offline one-time preprocessing step is already common, and since rewriting often reduces file size, it introduces no additional storage overhead. The closest related work, the proprietary Microsoft V-Order [26, 27, 37], reports an additional rewriting cost of 10-20% of the original write time and targets only Parquet configurations for CPU-based systems.

Future File Formats for GPUs. Several GPU-friendly encodings [1, 2, 13, 39] have been proposed to remove data dependencies to make decoding highly parallelizable on GPUs. These schemes report extremely high in-GPU-memory decoding throughput – up to 5700 GB/s [19, 40]. While these results are impressive, many real-world settings require reading files first from SSDs, and the dominant bottleneck is still storage I/O rather than computation [8, 14, 16, 24, 28, 42]. There is also a strong interest in new file formats across both academia and industry. Some efforts aim to replace Parquet with new file formats, mainly to support these new GPU-friendly encodings [19, 31, 40]. At the same time, the Apache Parquet community is discussing adding these encodings [3, 6, 33].

In this context, we present three perspectives on GPU file formats loaded from storage. First, for I/O-bound workloads, the extremely high in-memory decoding bandwidth offered by these encodings may provide limited benefit, since storage bandwidth is usually only on the order of tens of GB/s, far below GPU HBM bandwidth in the TB/s range. As a result, file scans are typically limited by I/O rather than by memory bandwidth or computation. Second, compression ratio matters in I/O-bound cases, as it directly improves effective bandwidth (see Figure 3). Finally, even if a new file format were introduced, much of the effort in this work would still be required in a similar form, namely, a rewriter from Parquet to the new file format. Doing so would also sacrifice the benefits of the Parquet ecosystem, especially compatibility and interoperability.

Future I/O Data Paths. One often overlooked dimension in studying file formats is the I/O data path, namely, where the input data is stored and how it is transferred. In this work, we use local-attached NVMe SSDs, since this setup is the simplest and also enables us to scale the number of SSDs. However, this is no longer the dominant configuration in the AI era, where cloud object storage [12] and network-attached storage [25] accessed via NICs are becoming increasingly important for GPUs. This shift is also visible in NVIDIA's high-end DGX servers integrating eight GPUs, each with its own NIC, but only a single SSD [29, 30]. We therefore expect future file-format studies to increasingly consider these storage settings.

Rewriting Beyond GPU. While we focus on GPUs, our rewriter is not GPU specific, and applies Parquet configurations without hardware-specific assumptions. Rewriting Parquet for different hardware simply means selecting different configurations. For example, Insight 3 and Insight 4 also apply to CPU-based systems, and parts of these ideas are already being discussed in the Arrow Rust library [4, 5, 15, 22, 41]. Overall, this leads to a simple principle: when a file is queried frequently, rewriting it into a hardware-optimized format can improve performance in subsequent queries. While this reasoning justifies new formats, we have shown it also applies to the Parquet format itself with a different configuration.

Acknowledgments

This work has been partially funded by the LOEWE Spitzenprofessur of the state of Hesse (III 5-519/05.00.003-(0005)), and 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 and grateful to Nils Boeschen, Andrew Lamb, and the RAPIDS team for their feedback.

References

[1] Azim Afroozeh and Peter Boncz. 2025. The FastLanes File Format. *Proc. VLDB Endow.* 18, 11 (2025), 4629–4643. doi:10.14778/3749646.3749718

[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*, Carsten Binnig and Nesime Tatbul (Eds.). ACM, 8:1–8:11. doi:10.1145/3662010.3663450

[3] Andrew Lamb and NVIDIA RAPIDS Development Team. 2026. cuDF Issue: Invitation to provide feedback on New Parquet Encoding: Adaptive Lossless Floating Point Compression. https://github.com/rapidsai/cudf/issues/21173. Accessed: 2026-04-19.

[4] Andrew Lamb and Xuwei Fu and Jigao Luo. 2025. Apache Arrow Rust Issue: [Parquet] Allow sampling some values to decide the encoding pattern. https://github.com/apache/arrow-rs/issues/8378. Accessed: 2026-02-01.

[5] Andrew Lamb and Xuwei Fu and Jigao Luo. 2025. Apache Arrow Rust Issue: Tuning knobs to tradeoff CPU and compression in parquet. https://github.com/apache/arrow-rs/issues/8358. Accessed: 2026-02-01.

[6] Apache Parquet Contributors. 2026. Parquet: Specification for FSST Encoding. https://docs.google.com/document/d/1Xg2b8HR19QnI3nhtQUDWZJhCLwJzW6y9tU1ziiLFZrM/. Accessed: 2026-04-19.

[7] 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). arXiv:2508.05029 doi:10.48550/ARXIV.2508.05029

[8] 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 (2024), 237:1–237:26. doi:10.1145/3698812

[9] DuckDB Documentation. 2026. DuckDB Parquet Tips. https://duckdb.org/docs/stable/data/parquet/tips. Accessed: 2026-02-01.

[10] DuckDB Documentation. 2026. DuckDB Performance Guide: File Formats. https://duckdb.org/docs/stable/guides/performance/file_formats. Accessed: 2026-02-01.

[11] NVIDIA Documentation. 2025. NVIDIA GPUDirect Storage. https://docs.nvidia.com/gpudirect-storage/

[12] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proc. VLDB Endow.* 16, 11 (2023), 2769–2782. doi:10.14778/3611479.3611486

[13] 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. doi:10.1145/3736227.3736242

[14] Yi Jiang, Hamish Nicholson, Viktor Sanca, and Anastasia Ailamaki. 2026. Data Movement-Aware GPU Sharing for Data-Intensive Systems. 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/data-movement-aware-gpu-sharing-for-data-intensive-systems.html

[15] Jörn Horstmann and Andrew Lamb and Xiangpeng Hao. 2025. Apache Arrow Rust Issue: Parquet LevelEncoder is much too eager to write short rle runs. https://github.com/apache/arrow-rs/issues/7739. Accessed: 2026-02-01.

[16] Joshua Patterson and Todd Mostak. 2026. NVIDIA GTC 2026: The Era of GPU Data Processing: From SQL to Search and Back Again. https://www.nvidia.com/en-us/on-demand/session/gtc26-s81769/. Accessed: 2026-04-19.

[17] 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: 2026-02-01.

[18] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2 (2023), 118:1–118:26. doi:10.1145/3589263

[19] Andrew Lamb and Julien Le Dem. 2026. Talk: Column Storage for the AI Era.

[20] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 5–17. doi:10.1145/3626246.3653368

[21] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 311–326. doi:10.1145/2882903.2882925

[22] Leonardo Yvens and Ed Seidl. 2025. Apache Arrow Rust Pull Request: Parquet: configurable data page v2 compression threshold. https://github.com/apache/arrow-rs/pull/9826. Accessed: 2026-04-30.

[23] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A Deep Dive into Common Open Formats for Analytical DBMSs. *Proc. VLDB Endow.* 16, 11 (2023), 3044–3056. doi:10.14778/3611479.3611507

[24] Jigao Luo, Nils Boeschen, Muhammad El-Hindi, and Carsten Binnig. 2025. PystachIO: Efficient Distributed GPU Query Processing with PyTorch over Fast Networks & Fast Storage. *CoRR* abs/2512.02862 (2025). arXiv:2512.02862 doi:10.48550/ARXIV.2512.02862

[25] Jigao Luo, Nils Boeschen, Tobias Ziegler, and Carsten Binnig. 2025. An Evaluation of NVMe-over-Fabrics for Disaggregated Databases over Fast Networks. In *Datenbanksysteme für Business, Technologie und Web (BTW 2025), 21. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 03.-07, März 2025, Bamberg, Germany, Proceedings (LNI)*, Meike Klettke, Ralf Schenkel, Andreas Henrich, Daniela Nicklas, Maximilian E. Schüle, and Klaus Meyer-Wegener (Eds.). Gesellschaft für Informatik e.V., 113–125. doi:10.18420/BTW2025-05

[26] Microsoft Fabric Warehouse Team. 2024. Understand V-Order for Microsoft Fabric Warehouse. https://learn.microsoft.com/en-us/fabric/data-warehouse/v-order. Accessed: 2026-02-01.

[27] Miles Cole. 2024. To V-Order or Not: Making the Case for Selective Use of V-Order in Fabric Spark. https://milescole.dev/data-engineering/2024/09/17/To-V-Order-or-Not.html. Accessed: 2026-02-01.

[28] 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. doi:10.1145/3736227.3736240

[29] NVIDIA Documentation. 2026. Introduction to NVIDIA DGX B300 Systems. https://docs.nvidia.com/dgx/dgxb300-user-guide/introduction-to-dgxb300.html. Accessed: 2026-04-19.

[30] NVIDIA Documentation. 2026. Introduction to NVIDIA DGX H100/H200 Systems. https://docs.nvidia.com/dgx/dgxh100-user-guide/introduction-to-dgxh100.html. Accessed: 2026-04-19.

[31] 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). arXiv:2504.15247 doi:10.48550/ARXIV.2504.15247

[32] Pedro Pedreira, Orri Erling, Maria Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proc. VLDB Endow.* 15, 12 (2022), 3372–3384. doi:10.14778/3554821.3554829

[33] Prateek Gaur and Apache Parquet Contributors. 2026. Apache Parquet Format Pull Request: Add ALP (Adaptive Lossless floating-Point) encoding specification. https://github.com/apache/parquet-format/pull/557. Accessed: 2026-04-19.

[34] 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, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1981–1984. doi:10.1145/3299869.3320212

[35] NVIDIA RAPIDS Development Team. 2025. NVIDIA RAPIDS libcudf, pylibcudf and cuDF: GPU DataFrame Library. https://github.com/rapidsai/cudf

[36] Karl B. Torp, Simon A. F. Lund, and Pinar Tözün. 2025. Path to GPU-Initiated I/O for Data-Intensive Systems. In *Proceedings of the 21st International Workshop on Data Management on New Hardware, DaMoN 2025, Berlin, Germany, June 22-27, 2025*. ACM, 3:1–3:9. doi:10.1145/3736227.3736232

[37] Uday Kumar Reddy Kallem. 2024. Direct Lake Performance — V-Order vs Default Write vs Custom Ordered Write. https://medium.com/@uday1409/direct-lake-performance-v-order-vs-default-write-vs-custom-ordered-write-7d34901724de. Accessed: 2026-02-01.

[38] Deepak Vohra. 2016. *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools* (1st ed.). Apress, USA. doi:10.1007/978-1-4842-2199-0

[39] Robin Vonk, Joost Hoozemans, and Zaid Al-Ars. 2025. GSST: Parallel string decompression at 191 GB/s on GPU. *ACM SIGOPS Oper. Syst. Rev.* 59, 1 (2025), 55–61. doi:10.1145/3759441.3759450

[40] Will Manning. 2025. FutureData: Vortex: LLVM for File Formats. https://db.cs.cmu.edu/events/futuredata-vortex/. Accessed: 2026-02-01.

[41] Xuwei Fu. 2025. Apache Arrow Rust Pull Request: Parquet: Do not compress v2 data page when compress is bad quality. https://github.com/apache/arrow-rs/pull/8257. Accessed: 2026-02-01.

[42] Gwangoo Yeo, Zhiyang Shen, Wei Cui, Matteo Interlandi, Rathijit Sen, Bailu Ding, Qi Chen, and Minsoo Rhu. 2026. ZipFlow: a Compiler-based Framework to Unleash Compressed Data Movement for Modern GPUs. *CoRR* abs/2602.08190 (2026). arXiv:2602.08190 doi:10.48550/ARXIV.2602.08190

[43] 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://www.cidrdb.org/cidr2026/keynotes.html

[44] Yichao Yuan, Advait Iyer, Lin Ma, and Nishil Talati. 2024. Vortex: Overcoming Memory Capacity Limitations in GPU-Accelerated Large-Scale Data Analytics. *Proc. VLDB Endow.* 18, 4 (2024), 1250–1263. doi:10.14778/3717755.3717780

[45] 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. doi:10.14778/3626292.3626298

[46] Xinyu Zeng, Ruijun Meng, Martin Prammer, Wes McKinney, Jignesh M. Patel, Andrew Pavlo, and Huanchen Zhang. 2025. F3: The Open-Source Data File Format for the Future. *Proc. ACM Manag. Data* 3, 4 (2025), 245:1–245:27. doi:10.1145/3749163