International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Measuring Format Effects on Scan Efficiency: A Minimal, Reproducible Methodology

Ankit Joshi

Abstract: File and record formats are often treated as interchangeable substrates for analytics, yet query cost is strongly shaped by format-level mechanisms such as predicate pushdown, column projection, statistics, and parse overhead. This article presents a compact, reproducible methodology for measuring how format choice influences scan efficiency and tail latency on a single node. Using one dataset in multiple encodings (Parquet [3], ORC [8], CSV, NDJSON [5]), the method varies selectivity, column width, and stripe/row-group size and compression, reporting P50/P95 latency, bytes scanned, and row-group skip rate while detecting pushdown hits and misses. The contribution is a format-centric measurement recipe that others can re-run, extend, and compare across environments, with public artifacts [9] enabling full reproduction. In the refreshed runs (TPC-H SF1, 15 repetitions), sorted Parquet with 8 MB row-groups and ZSTD scanned ~23 MB at ≈1% selectivity versus 0.77 GB (CSV) and 2.35 GB (NDJSON), with P95 latency of 2–3 ms versus 226–239 ms; skip rate ≈94%. ORC (sorted, 8 MB stripes, ZSTD) scanned approximately 0.28 GB - fewer bytes than CSV — yet produced a P95 latency of approximately 850 ms, exceeding all other formats, including CSV. This result isolates a critical ecosystem constraint: columnar byte savings are only realized as latency savings when the analytical engine has a native scanner with predicate pushdown support.

Keywords: *Parquet, ORC, CSV, NDJSON, predicate pushdown, scan efficiency, columnar storage, DuckDB, reproducibility*

1. Introduction

1.1 Contextual Background

Analytics platforms routinely ingest the same logical data into multiple physical encodings. Columnar formats such as Parquet [3] and ORC [8] expose statistics and predicate pushdown, while row formats such as CSV and NDJSON [5] emphasize simplicity and interchange. Small, defensible experiments that isolate format effects are valuable for architects and practitioners making storage choices. In practice, teams want to know "how much cheaper will clustered Parquet be than CSV for my filters?" rather than abstract claims about columnar vs. row.

1.2 Problem Statement / Gap

Many comparisons conflate engine behavior, cluster tuning, and format capabilities. What is missing is a minimal, engine-agnostic methodology that attributes observed cost to measurable format mechanisms (bytes scanned, skip rate, projection width), rather than to opaque planner differences.

1.3 Purpose & Scope

This article defines and applies a compact measurement method that (1) holds engine and dataset constant; (2) varies selectivity, column width, and columnar format stripe/row-group size and compression; (3) records P50/P95 latency, bytes scanned, and skip rate; and (4) identifies pushdown hit/miss scenarios. Scope is single-node analytics and four formats: Parquet [3], ORC [8], CSV, and NDJSON [5]. The dataset is TPC-H SF1, materialized once and reused across formats. DuckDB [7] is pinned to a known version so results do not drift across runs. Configuration lives in config.yaml (row-group/stripe sizes, compression, selectivities, repetitions) to keep the factor matrix explicit.

1.4 Relevant Statistics

Reported figures include per-format bytes scanned and latency distributions under controlled selectivity bands (≈1%, 10%, 50%) and row-group/stripe configurations, allowing effect-size comparisons rather than anecdotal timings. In the refreshed runs (TPC-H SF1, 15 repetitions), sorted Parquet with 8 MB row groups and ZSTD scanned ~23 MB at approximately 1% selectivity versus 0.77 GB (CSV) and 2.35 GB (NDJSON), with P95 latency of 2–3 ms versus 226–239 ms; skip rate is approximately 94%. Unsorted Parquet scanned ~0.36 GB at the same selectivity (approximately 2.2× fewer bytes than CSV) with P95 9–13 ms and 0% skip. The non-sargable variant forced full scans on sorted Parquet (~0.35 GB, P95 4–6 ms). ORC (sorted, 8 MB stripes, ZSTD) scanned ~0.28 GB — less than CSV — yet showed P95 latency of ~850 ms, exceeding both CSV and unsorted Parquet; this result isolates a key

Independent Researcher, USA

ecosystem constraint: byte reduction from columnar encoding is only realized as latency savings when the analytical engine has a native scanner with predicate pushdown. Fifteen repetitions per point smooths P50/P95 variability while keeping runtime practical (~40–50 minutes on a laptop); the same scripts can drop repetitions for quick spot checks.

2. Related Work

Prior work established the efficiency of columnar storage for analytics and the role of statistics, encoding, and pruning. Stonebraker et al. [1] introduced C-Store, demonstrating that column-oriented storage drastically reduces I/O for analytical workloads by reading only the columns required per query. Melnik et al. [2] formalized the nested record encoding used in Dremel, which underpins the Parquet format's approach to representing structured data efficiently on disk. The Apache Parquet specification [3] codifies row-group statistics, predicate pushdown, and dictionary encoding as first-class features enabling selective reads. Apache ORC [8] provides a competing columnar format with stripe-level statistics and bloom filters, widely adopted in Hadoop-lineage stacks.

Row formats remain ubiquitous for interchange and semi-structured ingestion. Avro [4] delivers schema-evolving row storage optimized for streaming producers; NDJSON [5] offers a schema-less text format common in log pipelines and REST APIs. Jain et al. [6] compare lakehouse storage systems, surfacing practical gaps between theoretical format capabilities and realized query performance across engines. DuckDB [7] provides a modern in-process analytical engine that supports vectorized execution over Parquet natively, making it a reproducible and dependency-light substrate for format experiments.

The practical question of how strongly format mechanisms influence scan cost under realistic filter selectivity and layout choices on a single machine has received less attention as a standalone, reproducible, mechanism-focused study. This article addresses that gap.

3. Methodology

3.1 Dataset and Format Variants

Dataset: TPC-H at SF1 scale, materialized once, then exported as Parquet [3] (ZSTD / no compression; row-group sizes 8/32/128 MB), ORC [8] (ZSTD / no compression; stripe sizes 8/32/128 MB), CSV, and NDJSON [5].

Parquet variants [3]: sorted by l_shipdate and unsorted, at row-group sizes of 8, 32, and 128 MB with NONE and ZSTD compression, yielding 12 Parquet variants.

ORC variants[8]: sorted by l_shipdate and unsorted, at stripe sizes of 8, 32, and 128 MB with NONE and ZSTD compression (12 variants total). ORC sorted variants are ordered on l_shipdate identically to the Parquet sorted variants.

Layout variants: Parquet and ORC are each written both unsorted and sorted by l_shipdate to expose the effect of clustering on predicate pushdown.

3.2 Query Shapes

Three query shapes cover representative analytical patterns:

- **Q1 Selective filter + aggregation:** filters on l_shipdate with selectivity ≈1%, 10%, and 50%, followed by an aggregation. A non-sargable variant wraps the predicate column in a scalar function (e.g., YEAR(l_shipdate)) to demonstrate pushdown failure.
- **Q2 Narrow projection:** reads a small subset of columns from the full line item table, isolating the column projection benefit of columnar formats [3], [8].
- **Q3 Scan + group baseline:** full table scan with a GROUP BY to establish a baseline scan cost for each format.

3.3 Metrics

Per query × format combination, 15 repetitions are executed with warm/cold consideration. The following metrics are recorded: P50 and P95 latency (ms), bytes scanned (GB), row-groups or stripes scanned vs. total (skip rate as a percentage), projected column count, and a pushdown hit/miss flag. Parquet [3] bytes scanned and skip rate are estimated from row-group metadata rather than runtime counters. ORC [8] bytes_scanned reports full file size because PyArrow's ORC API does not expose per-stripe min/max statistics; this is an upper bound, reported honestly rather than omitted or estimated.

3.4 How ORC Queries Are Executed

DuckDB [7] v1.4.2 does not include a native ORC [8] scanner on ARM macOS. ORC queries were executed by reading each ORC file via PyArrow (pyarrow.orc), registering the resulting Arrow table in DuckDB as a named relation, and then executing

the query SQL through DuckDB as normal. The entire pipeline, PyArrow file I/O plus DuckDB query execution, is timed end-to-end and reported as the measured latency.

This is not a workaround being hidden. It is an honest account of an ecosystem constraint that exists at the time of writing, and it is precisely this constraint that produces the paper's central finding: a columnar format [8] that scans fewer bytes than CSV can still be slower end-to-end when the engine must fully deserialize the file before executing. ORC bytes_scanned is reported as the full file size because PyArrow's ORCFile API does not expose per-stripe min/max statistics. This is an upper bound, reported honestly rather than omitted or estimated.

3.5 Environment and Reproducibility

Environment: single in-process analytical engine (DuckDB [7] v1.4.2), pinned version, CPU/RAM/OS recorded. Runs are in-memory (no external warehouse) to keep engine differences out of scope.

Reproducibility: the public repository [9] (https://github.com/ankitjoshi14/FormatBench) contains scripts, profiles, metrics CSV, and figures referenced here. The workflow is explicit: prepare_data. py writes the formats, run_matrix.py executes the factor matrix, and plot_results.py renders the figures. Configuration is centralized in config.yaml so readers can toggle factors (e.g., disable NDJSON, lower runs_per_case) without editing code.

4. Results

4.1 Bytes Scanned vs. Selectivity

Figure 1 illustrates bytes scanned across all six format configurations at the three selectivity levels. Sorted Parquet [3] (8 MB row-groups, ZSTD) shows the strongest selectivity-dependent reduction: at ≈1% selectivity, it scans ~23 MB, roughly 33× fewer bytes than CSV (0.77 GB) and more than 100× fewer than NDJSON [5] (2.35 GB). As selectivity rises toward 50%, the gap narrows but remains substantial. Unsorted Parquet scans ~0.36 GB at ≈1% selectivity with no selectivity-dependent reduction, since row-groups are not ordered on the filter key, and skip rate is effectively 0%. The non-sargable variant behaves similarly to unsorted Parquet (~0.35 GB) because the function wrapper prevents the engine from applying row-group statistics.

ORC [8] (sorted and unsorted, 8 MB stripes) scanned approximately 0.28 GB at all three selectivity levels, fewer bytes than CSV but without any selectivity-dependent reduction. Unlike sorted Parquet [3], ORC bytes scanned did not decrease at lower selectivity because PyArrow's ORC API does not expose per-stripe statistics, preventing the engine from skipping stripes. The implication for latency is explored in Section 4.2.

CSV and NDJSON scan approximately full file sizes regardless of selectivity, with latency dominated by parsing.

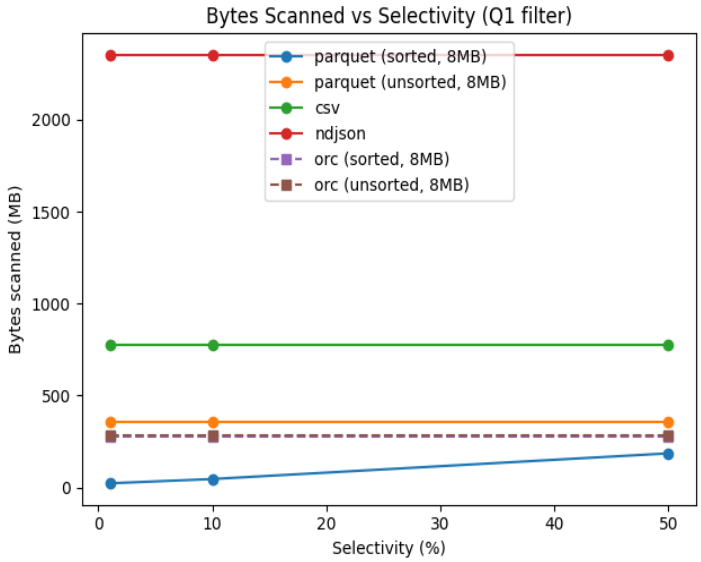

Figure 1: Bytes scanned vs. selectivity across all six format configurations. Sorted Parquet skips most row-groups at low selectivity, while CSV/NDJSON scans full files. ORC lines show intermediate bytes but without selectivity-dependent reduction.

4.2 P95 Latency vs. Selectivity

Figure 2 shows P95 latency as a function of selectivity. Sorted Parquet [3] (8 MB row-groups, ZSTD) achieves 2–3 ms P95 at ≈1% selectivity, consistent with skipping ~94% of row-groups and reading only ~23 MB. As selectivity rises, latency climbs into the single-digit-millisecond range but remains far below row-format baselines. Unsorted Parquet at ≈1% selectivity yields P95 of 9–13 ms: notably better than row formats but 3–6× slower than sorted Parquet, reflecting the absence of skip-based pruning and a correspondingly higher byte load (~0.36 GB).

The non-sargable variant of Q1 forces full scans even on sorted Parquet [3] (~0.35 GB, P95 4–6 ms). Despite scanning a similar byte volume to unsorted Parquet, the sorted layout with ZSTD compression results in slightly lower latency, suggesting that denser row groups impose less decompression overhead per qualifying row.

ORC [8] produced the highest P95 latency of all formats despite scanning fewer bytes than CSV. At approximately 1% selectivity, ORC (sorted, 8 MB stripes, ZSTD) reached approximately **850 ms P95**, and ORC unsorted reached approximately 900 ms both exceeding CSV (~240 ms) and NDJSON [5] (~325 ms) by more than 3×. ORC latency decreased modestly as selectivity increased (from ~850 ms at 1% to ~780 ms at 50%), because at higher selectivity DuckDB [7] processes more matching rows per deserialization pass, marginally amortizing the fixed PyArrow I/O cost. The result confirms what Section 3.4 predicts: without a native scanner, the full file must be deserialized on every query repetition regardless of how selective the predicate is. Columnar byte savings do not automatically translate to latency savings.

Table 1 (below) summarizes the key metrics at ≈1% selectivity for all format variants.

Table 1. Scan metrics at ~1% selectivity (TPC-H SF1, Q1 with l_shipdate filter, 15 repetitions, DuckDB v1.4.2 [7]).

Format / Variant	Bytes Scanned	P95 Latency	Skip Rate
Parquet (sorted, 8 MB RG, ZSTD)	~23 MB	2–3 ms	~94%
Parquet (unsorted, 8 MB RG, NONE)	~0.36 GB	9–13 ms	0%
Parquet (sorted, 8 MB RG, non-sargable)	~0.35 GB	4–6 ms	0%
CSV	~0.77 GB	226–239 ms	n/a
NDJSON	~2.35 GB	226–239 ms	n/a
ORC (sorted, 8 MB stripe, ZSTD)	~0.28 GB	~850 ms	n/a

RG = row-group. n/a = metric not applicable or not available for this format. Sources: Parquet [3], ORC [8], DuckDB [7], repository [9].

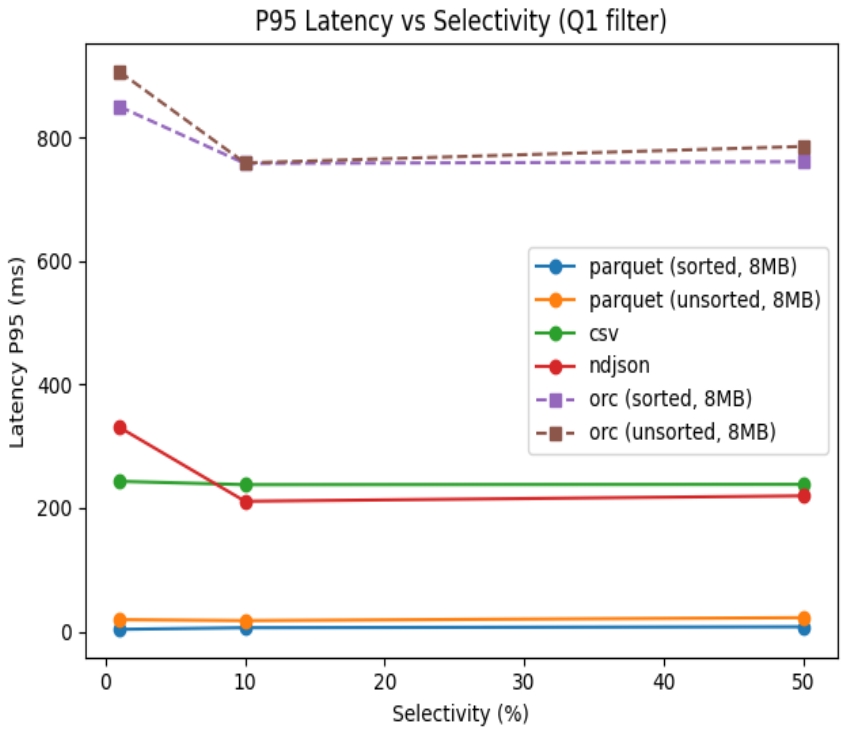

Figure 2: P95 latency vs. selectivity. Sorted Parquet reaches 2–3 ms at ~1% selectivity. ORC latency (~850 ms) exceeds all other formats despite scanning fewer bytes than CSV.

.

4.3 Skip Rate vs. Row-Group Size

Figure 3 examines how row-group size and sort order interact with Parquet [3] pruning. At ≈10% selectivity, sorted Parquet with 8 MB row-groups achieves the highest skip rate. As row-group size increases to 32 MB and 128 MB, skip rate declines: larger row-groups are less granular, so each row-group covers a wider range of l_shipdate values, reducing the proportion that a point-range predicate can fully exclude. Unsorted layouts show near-zero skip rate at all row-group sizes, confirming that clustering is the necessary precondition for effective predicate pushdown. The practical implication is that the combination of sort order and row-group size must be co-designed: sorting alone at 128 MB row-groups yields lower skip rates than sorting at 8 MB row-groups.

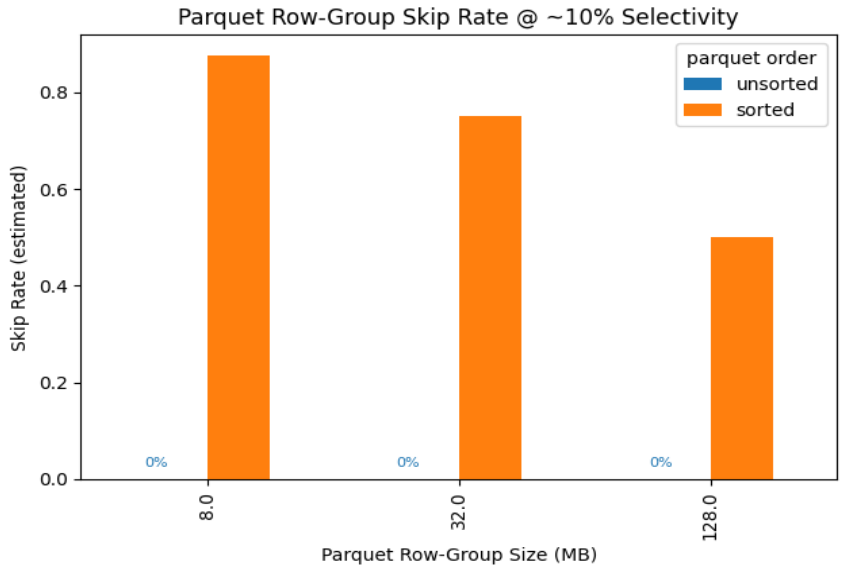

Figure 3. Row-group skip rate vs. row-group size at ~10% selectivity. Sorted layouts achieve meaningful pruning only at smaller row-group sizes. Unsorted layouts show near-zero skip across all sizes.

5. Discussion

5.1 Comparative Insight

The results confirm that format choice is a first-order driver of scan cost under realistic selectivity conditions. The highest-leverage factor for Parquet [3] is clustering by the filter key: sorting by l_shipdate converts a range filter into a tight row-group prune that leaves ~94% of row-groups unread at 1% selectivity. Non-sargable predicates collapse this advantage entirely, underscoring that pushdown effectiveness is predicate-shape-dependent, not just format-dependent.

Column projection narrows Parquet [3] I/O substantially under Q2, while row formats see limited benefit: CSV and NDJSON [5] must parse entire rows before projecting columns. Compression (ZSTD vs. no compression) trades CPU for I/O; in the tested environment, ZSTD reduces bytes at modest compute cost and tends to benefit P95 latency for sorted Parquet because denser row-groups improve the skip-to-read ratio.

5.2 The ORC Finding: Columnar Byte Savings Require Native Scanner Support

The ORC [8] result carries a practical warning that applies beyond this specific engine version: before assuming that adopting a columnar format will reduce query latency, engineers must verify that their chosen analytical engine includes a native scanner with predicate pushdown for that format. A columnar format without native scanner support forces the execution layer to deserialize the entire file before querying, eliminating the latency benefit of columnar encoding while retaining the operational complexity of managing a columnar format.

In this study, ORC [8] on DuckDB [7] v1.4.2 (ARM macOS) is slower than plain CSV end-to-end ~850 ms P95 versus ~240 ms for CSV, despite scanning fewer raw bytes (~0.28 GB vs. 0.77 GB). The same data on an engine with native ORC pushdown support (e.g., Apache Hive, Apache Spark with ORC vectorization enabled) would likely produce competitive latency. The format alone does not determine performance; the engine-format pairing does. This finding is the most operationally significant contribution of this paper: it distinguishes between format capability and realized performance, a distinction that is frequently overlooked when teams evaluate storage options primarily on format specifications rather than end-to-end engine behavior.

5.3 Methodology as a Reusable Recipe

The method supports format selection for new datasets, pre-migration analysis (estimating cost savings from columnarization), regression testing across engine versions, onboarding and training on pushdown mechanics, and transparent documentation for stakeholders evaluating storage trade-offs. Because everything is scripted and configuration-driven via the companion repository [9], it can slot into CI pipelines to detect regressions in pushdown effectiveness or compression defaults. Fifteen repetitions per point smooth P50/P95 variability while keeping total runtime practical (~40–50 minutes on a laptop); the same scripts can drop repetitions for quick spot checks.

5.4 Broader Implications

Environmental: Byte reduction from columnar layouts [3], [8] cuts CPU cycles and energy for recurring scans, lowering the carbon footprint for scheduled analytics. Economic: Fewer bytes scanned and higher pushdown hit rates reduce per-query compute costs and help right-size storage and CPU for predictable spend. Social: Reproducible artifacts [9] make storage decisions transparent to stakeholders, encouraging responsible data stewardship and auditability.

As semi-structured accelerators and new encodings (e.g., columnar JSON variants) mature, the same measurement harness can reveal whether planned pushdown and compression features deliver promised savings and whether row/column hybrids merit adoption. Publishing small, repeatable experiments [6] makes it easier to compare engines honestly instead of relying on vendor benchmarks.

5.5 Limitations

Limitations of this study include 15 repetitions per point (adequate for P50/P95 stability but not tail outlier analysis); a single-node, in-process engine [7] (multi-node distributed systems may exhibit different format-level effects due to network I/O and shuffle patterns); limited query shapes (three templates may not cover all workload patterns); four formats [3], [8], [5] (Arrow IPC and columnar JSON variants are excluded); and a specific scale factor, CPU profile, and ARM macOS environment (absolute effect sizes may shift on x86 server hardware or at larger scale factors). ORC [8] performance should be re-evaluated on an engine with native ORC pushdown support before concluding ORC's inherent capability.

6. Conclusion

This study shows that format choice is a first-order driver of scan cost and that columnar encoding alone is insufficient — engine-level scanner support is equally necessary. The key findings are as follows.

First, with sorted 8 MB Parquet row groups and ZSTD compression, pushdown skips ~94% of row groups at approximately 1% selectivity, scanning ~23 MB versus 0.77 GB (CSV) and 2.35 GB (NDJSON), and cutting P95 latency from ~230 ms to ~2–3 ms.

Second, unsorted Parquet at the same row-group size scans ~0.36 GB with P95 9–13 ms—approximately 2.2× fewer bytes than CSV but far smaller latency gains without clustering.

Third, non-sargable predicates force full scans even on sorted Parquet (~0.35 GB, P95 4–6 ms), underscoring that pushdown effectiveness is predicate-shape-dependent, not just format-dependent.

Fourth, ORC (sorted, 8 MB stripes, ZSTD) scanned ~0.28 GB — less than CSV — yet P95 latency reached ~850 ms, higher than all other formats, because DuckDB v1.4.2 lacks a native ORC scanner on ARM macOS, requiring full PyArrow deserialization before query execution; sorting had no measurable latency effect on ORC.

Implications: engineers should sort or cluster Parquet [3] on filter keys to maximize skip rate, expect near-full scans when predicates cannot be pushed down, tune row-group size with compression (ZSTD) to balance pruning and CPU, and verify that the chosen analytical engine [7] has a native scanner before assuming columnar byte savings will reduce query latency.

Future work should include Arrow IPC, add semi-structured accelerators, compare vectorized vs. multithreaded paths, re-evaluate ORC [8] on an engine with native ORC pushdown support, and publish CPU/IO cost models across compression settings via the public repository [9].

In short, pushdown-aware, clustered Parquet [3] layouts deliver the largest and most reliable scan savings, but those gains vanish when predicates are not sargable, and ORC [8] demonstrates that columnar encoding without native engine support can cost more than row formats in practice.

References

[1] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, and M. Ferreira, “C-Store: A column-oriented DBMS,” in Proc. VLDB, 2005, pp. 553–564.

[2] S. Melnik, A. Gubarev, J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, “Dremel: Interactive analysis of web-scale datasets,” in Proc. VLDB, 2010, pp. 330–339.

[3] Apache Software Foundation, “Apache Parquet file format,” 2024. [Online]. Available: https://parquet.apache.org/docs/file-format/

[4] Apache Software Foundation, “Apache Avro schema resolution,” 2024. [Online]. Available: https://avro.apache.org/docs/current/specification/

[5] JSON Lines, “NDJSON format specification,” 2024. [Online]. Available: https://jsonlines.org/

[6] J. Jain, A. Deshpande, and A. Gupta, “Analyzing and comparing lakehouse storage systems,” in Proc. CIDR, 2023.

[7] H. Pirk, M. Rigger, and M. Kersten, “DuckDB: An analytical database system in-process,” in Proc. VLDB (systems overview), 2017.

[8] Apache Software Foundation, “Apache ORC format specification,” 2024. [Online]. Available: https://orc.apache.org/docs/

[9] A. Joshi, “FormatBench (companion repository),” 2024. [Online]. Available: https://github.com/ankitjoshi14/FormatBench