write-direction

V4 Write-Direction Bridge Benchmarks

TL;DR

- Per-column content stats: `wrap` runs ~1.6–1.9x faster than per-row `convert`, which pays per-column allocation and eager bound decode.
- TrackedFile envelope: `convert` and `wrap` are within measurement error (9130 vs 8680 ops/ms) with null stats. Any small fixed edge doesn't scale with columns and is swamped once real column stats dominate the row.

Recommended shape: **one reusable adapter allocated per writer** and re-pointed per row, with **stats served by a map-backed view** (`MapBackedContentStats`) and `copy()` for stable snapshots. It is fastest or statistically tied at every column count with the lowest per-row allocation; the combined envelope-plus-stats result confirms wrap-both matches or beats convert-both and the hybrid.

Goal

These JMH microbenchmarks compare two strategies for presenting a legacy v2/v3 `ContentFile` (`DataFile`/`DeleteFile`) as a v4 manifest-entry row on the write path: `convert` (materialize a fresh struct per row) versus `wrap` (a reusable forwarding wrapper allocated once per writer and re-pointed at each file, with zero per-row allocation).

Methodology

Each benchmark measures the per-row cost of presenting one already-materialized legacy file as a v4 manifest-entry row — the work a manifest writer repeats for every entry it serializes.

The read side is held identical across variants: **every op walks the row positionally** through `StructLike` (`get(pos, Object.class)` at each ordinal, **descending into nested child structs**), **exactly as the writer does when serializing an entry**. Because that walk is the same for `convert` and `wrap`, the measured delta isolates per-row construction and allocation cost — not decoding or serialization I/O, and nothing is written to a file.

The per-row cost has two parts that scale differently and can move in opposite directions, so the work is split across three benchmarks — two that isolate each part and one that combines them:

- `ContentStatsBridgeBenchmark` measures content stats only — holding the envelope aside — and varies column count (2, 10, 50, 200) to capture the part that grows with table width.
- `TrackedFileBridgeBenchmark` leaves content stats null and captures only the fixed-size envelope (struct allocation plus array copies for split offsets, equality ids, and the nested tracking and partition structs).

`TrackedFileWithStatsBenchmark` measures both parts together — a full envelope plus per-column content stats in one row.

Isolating the parts keeps any wrapper win or loss attributable to a specific cost rather than an average of the two; the combined benchmark then shows how they net out in the shape a writer serializes.

Decode fairness is preserved so bound decoding stays out of the delta: `convert` decodes bounds **eagerly** when it builds the struct, `wrap` decodes **lazily** on first access during the read walk, and both therefore decode every bound exactly once per column per row. Schema-build fairness is preserved the same way: the per-column stats schema depends only on the table schema, not row data, so convert builds it once in setup — as a real writer would — rather than walking the schema per row.

JMH configuration: `@Fork(1), 3 warmup` iterations of 3s, 5 measurement iterations of 3s, throughput mode reported as operations per millisecond (`ops/ms`), higher is better. Allocation is measured with the GC profiler (`-prof gc`), reported as bytes per operation (`B/op`), lower is better.

ContentStatsBridgeBenchmark

Exposes a legacy file's five stat maps (`valueCounts`, `nullValueCounts`, `nanValueCounts`, `lowerBounds`, `upperBounds`) as v4 columnar stats, once per manifest entry, parameterized on column count (2, 10, 50, 200). Three variants:

- `convert` — materialize a real `ContentStatsStruct` plus a `FieldStatsStruct` per column every row, then read every field.
- `wrap` — the production `MapBackedContentStats`: a single reusable content wrapper re-pointed at the file's stat maps per row (zero per-row allocation).
- `wrapContentConvertFields` — a reusable content wrapper (no per-row content object), but each column's `FieldStats` is materialized as a fresh `FieldStatsStruct` on access. Isolates whether the reusable field view actually beats simply building a `FieldStatsStruct` per field.

numColumns	metric	convert	wrap	wrapContentConvertFields
2	ops/ms	5317.9 ± 2191.3	9953.6 ± 268.8	7294.8 ± 538.0
	B/op	1008	448	624
10	ops/ms	1189.4 ± 189.5	1899.2 ± 632.9	1354.0 ± 170.9

	B/op	3984	1664	3120
50	ops/ms	211.9 ± 20.2	384.2 ± 55.7	271.7 ± 77.0
	B/op	20704	7744	15600
200	ops/ms	44.4 ± 6.7	78.1 ± 13.2	56.8 ± 4.5
	B/op	98984	39280	76368

ops/ms rows: higher is better · B/op rows: lower is better

`wrap` runs ~1.6–1.9x faster than `convert` across all column counts once convert's stats-schema build is hoisted to setup, as a real convert writer would — the remaining gap is per-column allocation and eager bound decode, which the map-backed view avoids. Content stats production still dominates TrackedFile cost as column count grows. The hybrid variant lands at ~0.7x of `wrap`, confirming that the reusable map-backed field view (not just avoiding a per-row content object) is what delivers the win. Allocation tracks the same story: wrap allocates far less than convert (39,280 vs 98,984 B/op at 200 columns), and the bytes it does allocate are the per-column bound decode every variant shares, not object construction.

TrackedFileBridgeBenchmark

Presents a legacy `DataFile` as a v4 `TrackedFile` envelope (key metadata, split offsets, equality ids, nested tracking and partition structs), once per manifest entry. **Content stats are intentionally null here** — that cost is measured separately above — **so this isolates envelope construction** (struct allocation plus array copies). A single split offset is used; column count is irrelevant when stats are null.

variant	**ops/ms (higher is better)**	**B/op (lower is better)**
convert	9130.224 ± 1935.691	688
wrap	8679.966 ± 686.975	256

Here convert and wrap land within measurement error (9130 vs 8680 ops/ms). The envelope has a small, fixed field count, so the wrapper's per-access positional dispatch (a switch over ordinals) neither meaningfully helps nor hurts versus a one-shot struct copy. The wrap model's advantage shows only where per-row allocation scales with the data, i.e. content stats — though even here wrap allocates less (256 vs 688 B/op).

TrackedFileWithStatsBenchmark

Combines the two costs in one row: a full v4 TrackedFile envelope plus real per-column content stats, once per manifest entry, parameterized on column count (2, 10, 50, 200). This is the

shape a manifest writer serializes, so it shows the isolated costs above as they add up in practice.

- `convertBoth` — materialize a fresh `TrackedFileStruct` envelope and a fresh `ContentStatsStruct` (plus a `FieldStatsStruct` per column) every row.
- `wrapBoth` — the production reusable `DataTrackedFile`, allocated once and re-pointed per row; its stats are served by a reusable `MapBackedContentStats` view, so neither the envelope nor its stats allocate per row.
- `convertEnvelopeWrapStats` — the hybrid: a fresh envelope per row, but stats served by a reusable `MapBackedContentStats` re-pointed at the file rather than rebuilt.

numColumns	metric	convertBoth	wrapBoth	convertEnvelope WrapStats
2	ops/ms	2810.1 ± 97.1	3830.8 ± 279.2	3529.1 ± 42.1
	B/op	1648	704	1088
10	ops/ms	893.6 ± 27.6	1446.3 ± 35.6	1459.5 ± 18.8
	B/op	4624	1920	2304
50	ops/ms	186.8 ± 3.4	369.5 ± 7.4	360.0 ± 17.3
	B/op	21344	8000	8384
200	ops/ms	40.4 ± 0.6	71.5 ± 4.2	73.7 ± 3.9
	B/op	99624	39536	39920

ops/ms rows: higher is better · B/op rows: lower is better

`convertBoth` is ~1.4–2.0x slower than `wrapBoth` across all column counts. `wrapBoth` and `convertEnvelopeWrapStats` land within measurement error of each other, and neither consistently wins. They share the identical wrapped-stats path, the cost that dominates and scales, so they differ only in the envelope; the converted envelope that was tied in the isolated envelope benchmark gains nothing here, because that edge is a small fixed overhead swamped once per-column stats dominate the row, while converting the envelope still costs a per-row allocation that wrapping avoids. The combined result confirms the isolated findings: wrapping stats is what matters, and wrap-both is fastest or tied everywhere and allocates the least: the bytes it shows are the per-column bound decode every variant shares, while converting the envelope adds a flat ~400 B/op that does not scale with column count (see Table 3's B/op column).

Source

`ContentStatsBridgeBenchmark`:
https://gist.github.com/stevenzwu/aaee7dd8b090bf6a76a4f5eb4e571a5c

`TrackedFileBridgeBenchmark`:
https://gist.github.com/stevenzwu/deda788ac393b65d493dd4c2c847a95a

`TrackedFileWithStatsBenchmark`:
https://gist.github.com/stevenzwu/85016b55c3aede169ad651fd7fc76b6a

read-direction

V4 Read-Direction Bridge Benchmark

TL;DR

`wrap` runs ~1.3–1.9x faster than `convert` across all column counts and **allocates ~1.8–2.7x less**, so a read-path `TrackedContentFile` should present v4 `ContentStats` through lazy stat-map views instead of materializing five maps per row.

Boxing the `FieldStats` counts (`long` vs `Long`) gives no dependable win on the production `wrap` path — a narrow-table win (≤20 columns) that flips to mid-width regressions — so the counts stay `long`.

Methodology

The benchmark deserializes one manifest entry's v4 `ContentStats` once, then presents it through the legacy five-map `ContentFile` stat API (`valueCounts`, `nullValueCounts`, `nanValueCounts`, `lowerBounds`, `upperBounds`) on every op — the direction a scan-planning metrics evaluator reads a `TrackedContentFile`.

The read walk is held identical across variants: every op probes all five stats for every column by field id, matching how a metrics evaluator reads per-column bounds and counts. All variants re-encode bounds via `Conversions.toByteBuffer` on the same access, so the measured delta isolates map materialization versus a lazy view, not the encode work all variants share.

JMH: `@Fork(10), 3 warmup` iterations of 3s, 5 measurement iterations of 3s, Throughput in `ops/ms`, GC allocation profiler enabled. The schema rotates eight column types (boolean, int, `long`, float, double, string, date, timestamp) so bound encoding is polymorphic. `convert` and `wrap` are each measured twice — with `FieldStats` returning the three counts as primitive `long` (production) or boxed `Long` — giving the four table columns.

ContentStatsReadBenchmark

Presents a v4 manifest entry's `ContentStats` as the legacy five-map `ContentFile` stat API, then reads every column's five stats by field id.

`convert` — materialize five real `Maps` (pre-sized with the field count to avoid rehashing) from the content stats every op, then read every column.

`wrap` — five reusable `ContentStatsBackedMap` views backed directly by the content stats, computing each entry lazily on `get(fieldId)` with no per-op map allocation, then read every column.

The boxed variant tests whether the read path's per-access count boxing can be removed. `FieldStatsStruct` already stores the three counts as nullable `Long`, but the `FieldStats` interface returns primitive `long`, so `ContentStatsBackedMap` re-boxes each count with `Long.valueOf` on every `get(fieldId)` — an unbox-then-rebox round trip plus a fresh

`Long`. To verify the boxing/unboxing overhead hypothesis, the `FieldStats` interface was updated to return boxed `Long` directly:

```
long → Long valueCount();
long → Long nullValueCount();
long → Long nanValueCount();
```

Running each strategy against both count return types gives the four columns: **convert long** and **wrap long** use primitive `long` (production); **convert boxed** and **wrap boxed** use the boxed `Long` above.

numCol umns	metric	**convert long**	**wrap long**	convert boxed	wrap boxed
2	ops/ms	9710.1 ± 103.3	18259.5 ± 967.8	9953.4 ± 111.8	20667.2 ± 217.9
	B/op	1168	428	1120	352
10	ops/ms	1796.4 ± 14.6	2754.2 ± 21.9	1859.6 ± 23.5	3295.1 ± 40.2
	B/op	4864	2144	4624	1904
20	ops/ms	848.0 ± 10.4	1282.3 ± 21.8	876.8 ± 12.0	1574.1 ± 24.2
	B/op	9368	4288	8888	3808
50	ops/ms	320.1 ± 5.2	504.8 ± 6.0	344.8 ± 3.0	439.2 ± 38.6
	B/op	24864	10864	23664	9776
100	ops/ms	153.5 ± 1.2	243.6 ± 5.3	159.1 ± 2.1	190.0 ± 24.0
	B/op	49368	21728	46968	19496
200	ops/ms	64.7 ± 1.7	83.1 ± 1.3	70.8 ± 1.7	75.9 ± 1.0
	B/op	112200	56144	105648	49592
500	ops/ms	22.9 ± 0.9	35.4 ± 0.4	26.7 ± 0.9	37.1 ± 0.6
	B/op	305408	171872	284456	150920

o`ps/ms rows: higher is better · B/op rows: lower is better`

`wrap` wins because `convert` pays per-column map puts and eager bound encode on every op that the lazy view defers to actual `get(fieldId)` calls — running ~1.3–1.9x faster and allocating ~1.8–2.7x less across all column counts. `wrap` is optimal for the `get(fieldId)` access a metrics evaluator uses; an iteration-heavy `entrySet()` consumer would still materialize.

Boxing the `FieldStats` count returns (`long` vs `Long`) helps the eager `convert` path (+3–17%) but gives no dependable win on the production `wrap` path — a win only at narrow tables (up to +23% at ≤20 columns) that flips to the worst regressions mid-width (-13% at 50, -22% at 100, -9% at 200) — so counts stay `long`.

ContentStatsMapNullabilityBenchmark

Measures the cost of restoring the legacy null-when-empty contract on `ContentStatsBackedMap`: whether a metric no column tracks (for example, `nan_value_count` for a table with no floating-point columns) should present as `null`, matching the eager converters it replaced, or as a non-null empty map. Null-when-empty is what ships, via the `forKind` factory.

`emptyMap` — return each of the five `ContentStatsBackedMap` views directly; a metric no column tracks presents as a non-null empty map.

`nullViaFactory — an allocation-free` emptiness scan per view, short-circuiting on the first column that contributes an entry, returning `null` instead of an empty map when none do — matching the legacy contract; this is the production `forKind` path.

numColumns	emptyMap	nullViaFactory	vs wrap-long read
2	127324.0 ± 3746.7	36749.9 ± 931.7	50% of read
10	128965.4 ± 1120.8	34779.2 ± 865.9	8% of read
50	128770.6 ± 1677.8	35055.5 ± 509.9	1.4% of read
200	125289.6 ± 4614.2	28672.4 ± 3736.6	0.3% of read

`ops/ms` (higher is better)

`emptyMap` only allocates the five views — flat at ~127K `ops/ms` — so it is a do-nothing floor, not a meaningful cost baseline. `nullViaFactory` adds the `forKind` emptiness scan and stays flat too (~30–37K `ops/ms`): every metric short-circuits on an early contributing column — value and null counts and bounds on the first column, `nan_value_count` on the first float/double column — so no scan walks the whole schema. The meaningful yardstick is the read the scan precedes: against `ContentStatsReadBenchmark`'s `wrap` long, the scan is ~50% of the read at 2 columns and falls to ~0.3% at 200 (`wrap` long: 18,259 → 83 `ops/ms`) — a shrinking sliver of the work every entry does anyway.

Decision: ship null-when-empty (Option B), via the `forKind` factory. It restores the legacy contract — the eager converters returned null when no column tracks a metric, and scan-planning code branches on that null, so a non-null empty map would be a silent behavioral

change. The scan that buys it back short-circuits for every tracked metric and, against the actual per-entry read, costs under 1.5% at any realistic table width — so the correctness win is essentially free.

Source

`ContentStatsReadBenchmark`:
https://gist.github.com/stevenzwu/337d5dfeb5170d0b75df91dd111c1970

`ContentStatsMapNullabilityBenchmark`:
https://gist.github.com/stevenzwu/a338bc205a536d049fdcf1ee02be0085