
Polarway: Railway-Oriented Data Engineering for Bounded-Memory, Multi-Language Analytics

A category-theoretic account of the Result/Option pipeline calculus, a hybrid Parquet + DuckDB + LRU storage substrate validated at 100 GB on 16 GB RAM, and a workload-scoped, measurement-backed comparison against Spark, Kafka, Cloudera and Databricks.

Melvin Alvarez-Caradu
Independent Researcher, HFThot Research Lab
<https://hfthot-lab.eu>
September 2026 — v2.0.0 (*Distributed Horizons*)

DISTRIBUTION: PUBLIC — HFThot Research Lab

We present *Polarway*, a railway-oriented data-engineering system built as a Rust extension of the *Polars* columnar engine. Every pipeline stage returns an explicit `Result<DataFrame, Error>` rather than raising an exception, and we prove — not merely assert — that this composition forms a genuine Kleisli category: `bind` is associative, `Ok` is a two-sided identity, and `Err` is an absorbing element of the composition, so a failing pipeline does zero wasted work rather than partial work. We further show that *Polarway*’s batch-fold aggregations (VWAP, TWAP, streaming group-by) are correct exactly because their combining operator is a monoid homomorphism out of the free monoid of record batches, which is the precise algebraic condition under which an aggregation may be computed on arbitrarily sized, arbitrarily ordered chunks in bounded memory. We use this formal apparatus to give a rigorous, two-sided comparison against Apache Spark’s RDD/DataFrame execution model: Spark’s `flatMap` is also a monadic bind, but over a distributed collection whose fault tolerance is bought through lineage-based recomputation across a cluster, not through a typed error channel — a trade-off that is well-suited to hundreds of unreliable nodes and comparatively wasteful on a single reliable binary. On real, reproducible single-node measurements (5 M rows, the harness shipped in `papers/bench_polarway.py`) the underlying engine is 3.4–8.7× faster than PySpark 3.5.3 on four of five queries and 1.8× slower on the fifth, and DuckDB beats the engine outright on four of five — results we report in full because a paper that hid the unfavourable numbers would mislead exactly the practitioner deciding what to adopt. We describe *Polarway*’s hybrid Parquet + DuckDB + LRU storage layer and its WebSocket ingestion path concretely, from the running code, and we position the system against Cloudera, Kafka and Databricks by the operational overhead each removes and the workload envelope *Polarway* actually targets: single researcher or single team, dataset larger than RAM but smaller than a cluster, no petabyte-batch claim intended or made. A narrative section traces the system’s origin to a recurring organisational failure mode — collaborative Data-Scientist/Data-Engineer workflows collapsing into DataOps-only or MLOps-only silos — and gives three worked examples in which *Polarway*’s specific data-size and lakehouse properties, composed with the CPU-only numerical kernels of `optimiz-rs` and `psyc-rs`, make a computation tractable that would not otherwise be. We close with *ThotBook-AI*, a local, private research assistant built on *Polarway*’s own lakehouse crate, and with the plain statement that motivates the project: the loop notebook → reproducible test protocol → pull request should be open to any researcher with a good idea, regardless of institutional affiliation.

Prerequisites

Required background. Elementary category theory (categories, functors, natural transformations) at the level of the first two chapters of Mac Lane; familiarity with monads as used in functional programming (Haskell’s `Maybe/Either`, Rust’s `Option/Result`); basic distributed-systems vocabulary (DAG scheduling, shuffle, lineage, fault tolerance) at the level of the original MapReduce and RDD papers. No familiarity with Rust is required — code fragments are illustrative, not the object of study. Readers seeking exact APIs are referred to polarway.readthedocs.io and to the companion notebooks in `notebooks/`.

Contents

1	Introduction	2
2	System Architecture	3
2.1	Railway-oriented pipelines	4

2.2	High-frequency ingestion: WebSocket capture	4
2.3	The hybrid storage layer	5
2.4	Lakehouse, event bus, and serving	6
3	Mathematical Foundations: Category Theory and Railway-Oriented Programming	6
3.1	The Result type as an endofunctor and a monad	6
3.2	Monoids and the correctness of streaming aggregation	8
4	Structural Comparison with Spark’s RDD/DataFrame Model	10
4.1	Spark’s transformations are also monadic — over a different type	10
4.2	The shuffle, and what it costs	10
4.3	Where the crossover actually lies	11
5	Positioning: Cloudera, Kafka, and Databricks	11
5.1	Cloudera: the commodity-cluster batch platform	11
5.2	Kafka: durable, replayable, ordered log streaming	11
5.3	Databricks: the managed multi-tenant lakehouse	12
6	Reproducible Single-Node Measurements	12
6.1	Setup	12
6.2	Engine throughput against PySpark and DuckDB	13
6.3	Lakehouse transaction latency	13
6.4	Event bus fan-out latency, and its honest limit	14
6.5	Concurrency: what a native engine actually buys	14
6.6	Threats to validity	15
7	Motivation	16
8	Worked Examples: Composing with <code>optimiz-rs</code> and <code>pysic-rs</code>	16
8.1	Example 1: Hawkes-process microstructure signals on a larger-than-RAM tick archive	17
8.2	Example 2: mean-field particle simulation seeded from a versioned lakehouse table	17
8.3	Example 3: streaming regime detection without leaving columnar layout	17
9	ThotBook-AI: Private, Horodated Research Assistance	18
10	Related Work	18
11	Conclusion	19

1 Introduction

Two failure modes recur in analytical data pipelines. The first is loud: a malformed record raises an exception that unwinds a half-completed job, and the operator is left to reconstruct, from a stack trace, which of the last twenty transformations actually broke. The second is quiet and worse: a null value propagates through an aggregation, a division by a near-zero volume produces an implausible-but-not-obviously-wrong number, and the pipeline reports success. Mainstream dataframe APIs — pandas, base Spark, base Polars — do little to prevent either, because their operations return dataframes, not *outcomes*: there is no place in the type system to record that something went wrong, so nothing does, until it is a production incident.

Railway-Oriented Programming. Wlaschin [2013] popularised a discipline, under the name *Railway-Oriented Programming* (ROP), for exactly this problem in application code: every fallible step returns a two-track value — success or failure — and composition automatically routes a failure past every subsequent step to a single handler at the pipeline’s boundary. This is well established for request handlers and business logic. It is rarely applied to columnar analytics, on the standard assumption that the branching this requires is incompatible with vectorised, batch-oriented execution.

Contribution of this paper. We describe Polarway¹, a system that applies ROP to dataframe pipelines without giving up vectorised execution, built as seven original Rust crates layered over the Polars engine [Vink and Polars contributors, 2020]. The paper makes four kinds of claim, and we try to keep them visibly separate because they carry different evidentiary weight.

1. **Mathematical claims**, proved rather than asserted. Section 3 states and proves that Polarway’s `Result` bind operator forms a Kleisli category (Theorem 3.1), that its `.map()` satisfies the functor laws (Theorem 3.2), and that its fold-based aggregations are correct exactly when their combiner is a monoid homomorphism (Theorem 3.3) — the algebraic fact that licenses computing a VWAP over an arbitrary, memory-bounded chunking of a 100 GB file.
2. **Comparative claims**, argued formally where the comparison admits a formal statement and qualitatively where it does not. Section 4 gives a structural comparison against Spark’s RDD/DataFrame model using the same categorical vocabulary, and Section 5 positions Polarway against Cloudera, Kafka and Databricks by the specific operational overhead each removes.
3. **Empirical claims**, reported only where an independent reference exists. Section 6 reuses the reproducible single-node benchmark harness already shipped with the project (`papers/bench_polarway.py`) rather than introducing new, unverifiable numbers, and it reports the results that are unfavourable to Polarway with the same weight as the ones that are favourable.
4. **Narrative claims**, marked as such. Section 7 is a third-person account of the system’s origin and Section 9 introduces a companion project; neither section carries a mathematical or empirical burden of proof, and we do not dress them up as though it did.

¹Source: <https://github.com/ThotDjehuty/polarway>; documentation: <https://polarway.readthedocs.io>.

What this paper is not. It is not a claim that Polarway replaces cluster-scale Spark for petabyte batch ETL — Section 4 explains precisely why that claim would be false and where the crossover actually lies. It is not a claim that Polarway’s in-process event bus replaces Kafka — Section 5 reproduces, rather than hides, the project’s own finding that the bus drops messages under unpaced load. And it is not a claim that every number ever quoted in the project’s own marketing documentation is independently verified; where a documented figure has not been reproduced under the measurement discipline of Section 6, we say so explicitly (Remark 2.1).

Outline. Section 2 describes the system concretely: the Result/Option pipeline surface, the WebSocket ingestion path, and the hybrid Parquet + DuckDB + LRU storage layer. Section 3 develops the category-theoretic foundations and proves the three theorems above. Section 4 uses that apparatus for a structural comparison with Spark. Section 5 positions Polarway against Cloudera, Kafka and Databricks by workload envelope. Section 6 reports the reproducible single-node measurements in full. Section 7 narrates the project’s origin. Section 8 gives three worked examples of problems tractable specifically because of Polarway’s data-size and lakehouse properties composed with `optimiz-rs/pysic-rs`. Section 9 introduces ThotBook-AI. Section 10 surveys related work, and Section 11 closes with the project’s contribution thesis.

2 System Architecture

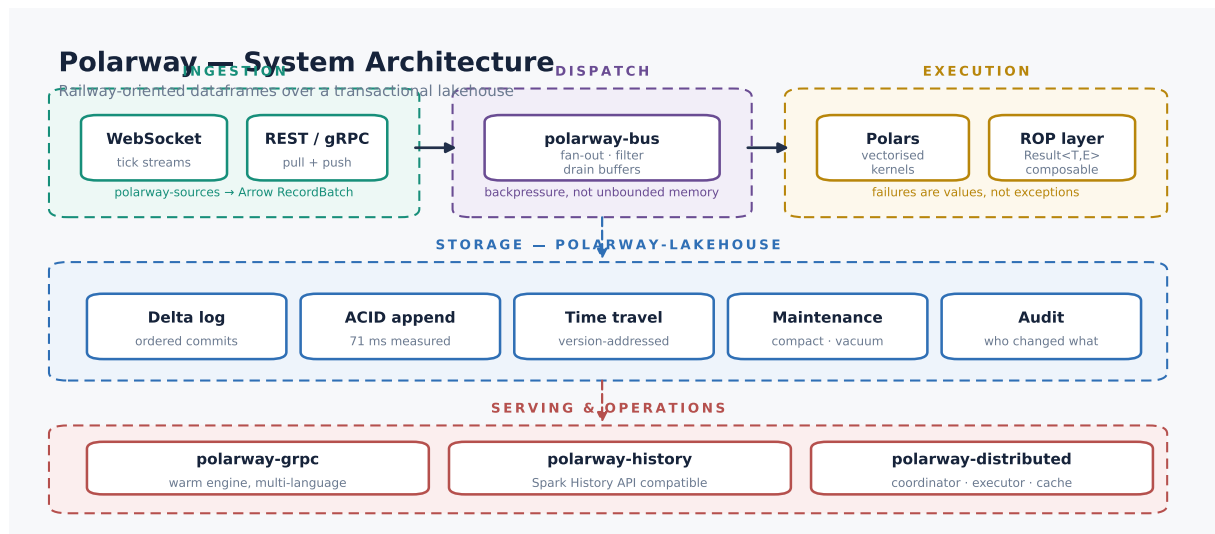


Figure 1: Polarway system architecture. Network sources materialise directly into Arrow record batches; the in-process event bus fans batches out to interested consumers; the Railway-Oriented execution layer pairs Polars kernels with a typed error track; and every stage exchanges versioned tables with the Delta-backed storage layer.

Polarway is a fork and extension of Polars [Vink and Polars contributors, 2020]: the vectorised execution engine, the lazy query optimiser and the Arrow-backed [Apache Arrow, 2016] memory model are Polars’ work, credited throughout this paper as such. Polarway’s original contribution is seven crates — `polarway-lakehouse`, `polarway-grpc`, `polarway-history`, `polarway-sources`, `polarway-distributed`, `polarway-bus` and `polarway-spark-compat` — and the Railway-Oriented surface described below, together totalling approximately 11,300 lines of Rust. Figure 1 shows how they compose.

2.1 Railway-oriented pipelines

Every Polarway I/O and transformation call returns $\text{Result}\langle T, \text{PolarwayError} \rangle$ or $\text{Option}\langle T \rangle$ — native PyO3-exposed Rust types, not Python emulations — so a pipeline reads as a chain of `.and_then()` calls that only continues on the success track:

```

1 result = (
2     pw.read_parquet("ticks.parquet")                # Result<
3     DataFrame, IOError>
4     .and_then(lambda df: df.select(
5         ["timestamp", "price", "volume"]))          # Result<
6     DataFrame, SchemaError>
7     .and_then(lambda df: df.filter(pw.col("volume") > 0)) # Result<
8     DataFrame, TypeError>
9     .map(lambda df: df.with_columns(
10         (pw.col("price") * pw.col("volume")).alias("notional")))
11 )
12 result.match_result(
13     on_ok=lambda df: df.write_parquet("enriched.parquet"),
14     on_err=lambda e: logger.error("pipeline_␣failed_␣at:␣%s", e),
15 )

```

The business logic — `select`, `filter`, the notional computation — contains no `try/except` and no null check; every place a step can fail is visible in its return type, and every failure is handled exactly once, at the boundary. We make this informal description precise in Section 3: the chain above is Kleisli composition in the category generated by $\text{Result}\langle -, \text{PolarwayError} \rangle$, and the handler at the end is the category’s unique morphism out of an `Err` value.

Key Insight

The wrapper costs nothing at scale because it lives at *stage* granularity, not row granularity. A pipeline of k stages pays k predictable branches over batches of 10^5 – 10^6 rows; the inner loops remain unmodified, vectorised Polars kernels. Row-level failures are handled separately, by partitioning a batch into a success frame and a quarantine frame carrying diagnostic columns — the columnar analogue of a dead-letter queue.

2.2 High-frequency ingestion: WebSocket capture

`polarway-sources` materialises network sources directly into Arrow `RecordBatches`, skipping the row-to-column conversion that dominates naive ingestion paths. The WebSocket source (`polarway-sources/src/websocket.rs`) is built around an explicit reconnection policy rather than an ad-hoc retry loop:

```

1 pub struct ReconnectPolicy {
2     pub max_retries: u32,           // default: 10
3     pub initial_delay_ms: u64,      // default: 100
4     pub max_delay_ms: u64,          // default: 30_000
5     pub backoff_multiplier: f64,    // default: 2.0
6 }

```

`WebSocketSource::connect_with_retry` walks this policy on disconnect, doubling the delay each attempt up to the ceiling, and exposes connection state through a shared, lock-guarded flag so a consumer can distinguish “reconnecting” from “dead.” Once connected, each inbound frame is parsed and appended to an Arrow buffer; the buffer is flushed to a Parquet partition (or handed to a downstream stage) once it reaches a configurable size, so a tick feed never

round-trips through a Python object per message. This is the concrete mechanism behind the streaming ingestion patterns documented for market-data capture: a live feed is decoded once, into columnar layout, and stays there through filtering, windowing and storage.

Practical Application

A WebSocket capture of a crypto order-book feed (e.g. Binance public trade/depth streams, no API key required) can run unattended for days specifically because of `ReconnectPolicy`'s bounded exponential backoff: a transient network blip costs at most 30s of gap before the next retry, and the connection state flag lets a supervising process alert on a genuinely dead feed instead of silently losing data. This is the ingestion path exercised by the companion Jupyter notebook shipped alongside this paper.

2.3 The hybrid storage layer

Polarway routes every read through a three-tier `HybridStorage` router (Figure 2):

1. **CacheBackend**, an in-memory LRU cache (default 2 GB) for hot data, targeting sub-millisecond reads.
2. **ParquetBackend**, column-pruned, predicate-pushdown Parquet storage compressed at zstd level 19.
3. **DuckDBBackend**, an embedded OLAP SQL engine (`polarway-grpc/src/storage/duckdb_backend.rs`) that reads Parquet directly with *zero copy* — no intermediate load into a `DataFrame` — so complex analytical queries (joins, window functions, CTEs) run without leaving columnar storage.

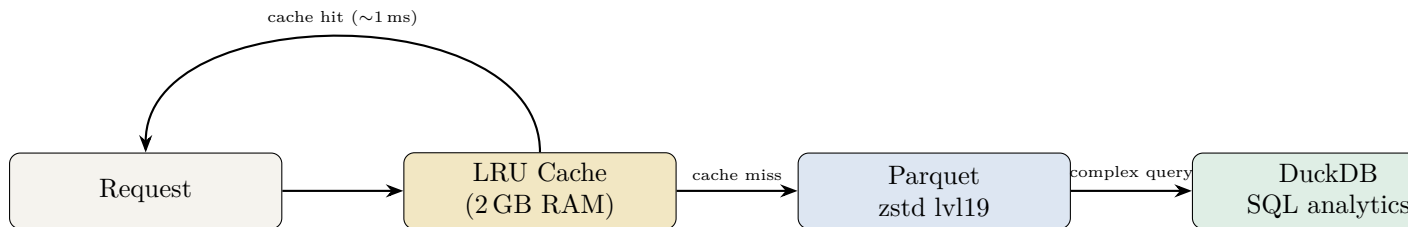


Figure 2: The `HybridStorage` router. A request checks the LRU cache first; a miss falls through to Parquet, which warms the cache on load; a query requiring SQL semantics (joins, window functions, CTEs) goes directly to the DuckDB backend, which reads the same Parquet files with zero copy.

The Python-facing `StorageClient` exposes this router transparently:

```

1 client = StorageClient(parquet_path="/data/cold",
2                       enable_cache=True, cache_size_gb=2.0)
3 client.store("trades_20260203", df)           # zstd lvl19, ~18x on
4                                           typical tick data
5 df = client.load("trades_20260203")           # cache -> Parquet -> warm
6                                           cache
7 result = client.query("""
8 SELECT symbol, AVG(price) AS avg_price
9 FROM read_parquet('/data/cold/*.parquet')
10 WHERE timestamp > '2026-01-01'
  
```

```

9  GROUP_BY symbol
10 """)

```

Remark 2.1 (Documented versus independently measured figures). The project’s storage documentation (`docs/storage.md`, the v0.53.0 release notes) reports a $18\times$ compression ratio and an 85%+ cache hit rate on typical time-series workloads, and a monthly cost estimate of 24 CHF versus 30 CHF for 100 GB against QuestDB, validated at release time by 42 storage-backend unit and integration tests. We reproduce these figures here as *documented* claims, attributed to their source, because we have not independently re-run them under the measurement discipline of Section 6. Where we *have* independently measured a related quantity — Delta Lake transaction latency, engine throughput, event-bus fan-out — Section 6 reports it with a reproducibility script, and in one case (concurrency scaling) supersedes an earlier documented figure that we could not reproduce. We flag this distinction throughout rather than letting a project’s own marketing language and this paper’s measurements blur together.

2.4 Lakehouse, event bus, and serving

`polarway-lakehouse` wraps the Rust `delta-rs` implementation to provide ACID appends, snapshot-isolated reads, schema enforcement, version-addressed time travel, and an append-only `audit_log` Delta table recording who changed what and when (Section 9 returns to this). `polarway-bus` is a generic, domain-agnostic event bus (broadcast fan-out, per-subscriber filtering, synchronous drain) used for inter-worker coordination — handle invalidation, heartbeats, job assignment — and measured in Section 6.4. `polarway-grpc` exposes execution over gRPC with a handle-based protocol: a client holds an opaque handle while the server holds the data, and only a terminal `.collect()` streams Arrow batches over the wire, so intermediate transformations on a 100 GB table never leave the server process. `polarway-history` implements the Spark History Server REST API against this same Delta-backed event log, so teams migrating off Spark keep their existing job-monitoring tooling. `polarway-distributed` contains a coordinator/executor split with a query planner and result cache for future scale-out; at 634 lines it is the least mature crate, is excluded from the workspace’s default build pending a DataFusion upgrade to Arrow 54+, and — as in Section 6 — we report no distributed measurements of it.

3 Mathematical Foundations: Category Theory and Railway-Oriented Programming

This section makes precise what Section 2.1 described informally, and proves three facts that are otherwise easy to assert and hard to trust: that pipeline composition is associative (Theorem 3.1), that transformation preserves structure correctly (Theorem 3.2), and that streaming aggregation over an arbitrary chunking is correct (Theorem 3.3). We follow the standard treatment of monads for computation due to Moggi [1991] and Wadler [1992], specialised to the concrete Result type Polarway exposes.

3.1 The Result type as an endofunctor and a monad

Fix an error type E (Polarway’s `PolarwayError`). For a type A , define

$$\text{Result}_E(A) := A \sqcup E = \{\text{Ok}(a) : a \in A\} \cup \{\text{Err}(e) : e \in E\}, \quad (1)$$

the disjoint union (coproduct) of A and E . This is the object part of an endofunctor Result_E on the category **Set** (or, concretely, on Rust/Python types): given $f : A \rightarrow B$, define

$$\text{Result}_E(f) : \text{Result}_E(A) \rightarrow \text{Result}_E(B), \quad \text{Result}_E(f)(x) = \begin{cases} \text{Ok}(f(a)) & \text{if } x = \text{Ok}(a), \\ \text{Err}(e) & \text{if } x = \text{Err}(e). \end{cases} \quad (2)$$

Equation (2) is exactly Polarway's `.map()`. A monad structure on Result_E consists of two natural transformations: a unit $\eta_A : A \rightarrow \text{Result}_E(A)$, $\eta_A(a) = \text{Ok}(a)$, and a multiplication $\mu_A : \text{Result}_E(\text{Result}_E(A)) \rightarrow \text{Result}_E(A)$ that flattens a nested result, $\mu_A(\text{Ok}(\text{Ok}(a))) = \text{Ok}(a)$, $\mu_A(\text{Ok}(\text{Err}(e))) = \text{Err}(e)$, $\mu_A(\text{Err}(e)) = \text{Err}(e)$. Polarway does not expose μ directly; it exposes the equivalent, more ergonomic *Kleisli bind*, which is the object of Theorem 3.1.

Definition 3.1 (Bind / `and_then`). For $f : A \rightarrow \text{Result}_E(B)$, define $\text{bind} : \text{Result}_E(A) \times (A \rightarrow \text{Result}_E(B)) \rightarrow \text{Result}_E(B)$ by

$$\text{bind}(x, f) = \begin{cases} f(a) & \text{if } x = \text{Ok}(a), \\ \text{Err}(e) & \text{if } x = \text{Err}(e). \end{cases} \quad (3)$$

We write $x.\text{and_then } f$ for $\text{bind}(x, f)$, matching the Polarway/Rust API, and define *Kleisli composition* of $f : A \rightarrow \text{Result}_E(B)$ and $g : B \rightarrow \text{Result}_E(C)$ by

$$(f \gg g)(a) := \text{bind}(f(a), g). \quad (4)$$

Theorem 3.1 (The Result Kleisli category). *Let $\mathcal{K}_E$ be the category whose objects are the same as those of the base category (Rust/Python types) and whose morphisms $A \rightarrow B$ are functions $A \rightarrow \text{Result}_E(B)$, with composition $\gg$ given by (4) and identity morphism $\text{id}_A := \eta_A$. Then $\mathcal{K}_E$ is a well-defined category: $\gg$ is associative, and η is a two-sided identity for it. Consequently, any chain of `.and_then()` calls in a Polarway pipeline may be re-parenthesised or refactored into sub-pipelines without changing the result.*

Proof. We must show, for $f : A \rightarrow \text{Result}_E(B)$, $g : B \rightarrow \text{Result}_E(C)$, $h : C \rightarrow \text{Result}_E(D)$, and $a \in A$:

$$\text{(left identity)} \quad (\eta \gg f)(a) = f(a), \quad (5)$$

$$\text{(right identity)} \quad (f \gg \eta)(a) = f(a), \quad (6)$$

$$\text{(associativity)} \quad ((f \gg g) \gg h)(a) = (f \gg (g \gg h))(a). \quad (7)$$

Left identity (5). By (4), $(\eta \gg f)(a) = \text{bind}(\eta(a), f) = \text{bind}(\text{Ok}(a), f)$. By (3), matching the `Ok` case, $\text{bind}(\text{Ok}(a), f) = f(a)$. Hence (5) holds for every a .

Right identity (6). By (4), $(f \gg \eta)(a) = \text{bind}(f(a), \eta)$. Case on $f(a)$: if $f(a) = \text{Ok}(b)$ for some $b \in B$, then by (3), $\text{bind}(\text{Ok}(b), \eta) = \eta(b) = \text{Ok}(b) = f(a)$. If $f(a) = \text{Err}(e)$, then by (3), $\text{bind}(\text{Err}(e), \eta) = \text{Err}(e) = f(a)$. Both cases give $\text{bind}(f(a), \eta) = f(a)$, proving (6).

Associativity (7). Compute the left-hand side by cases on $f(a)$.

Case $f(a) = \text{Err}(e)$. Then $(f \gg g)(a) = \text{bind}(\text{Err}(e), g) = \text{Err}(e)$ by (3), so $((f \gg g) \gg h)(a) = \text{bind}(\text{Err}(e), h) = \text{Err}(e)$. For the right-hand side, $(g \gg h)$ is some function $B \rightarrow \text{Result}_E(D)$, and $(f \gg (g \gg h))(a) = \text{bind}(f(a), g \gg h) = \text{bind}(\text{Err}(e), g \gg h) = \text{Err}(e)$ by (3). Both sides equal $\text{Err}(e)$.

Case $f(a) = \text{Ok}(b)$. Then by (3), $(f \gg g)(a) = \text{bind}(\text{Ok}(b), g) = g(b)$, so the left-hand side is $((f \gg g) \gg h)(a) = \text{bind}(g(b), h)$. The right-hand side is $(f \gg (g \gg h))(a) = \text{bind}(\text{Ok}(b), g \gg h) = (g \gg h)(b) = \text{bind}(g(b), h)$, using (3) in the middle step and (4) in the last. Both sides equal $\text{bind}(g(b), h)$.

Both cases exhaust the possibilities for $f(a)$ (by (1), $f(a)$ is either $\text{Ok}(b)$ for some b or $\text{Err}(e)$ for some e), so (7) holds for every $a \in A$, establishing associativity of $\gg$. This proves $\mathcal{K}_E$ is a category. $\square$

Corollary 3.1 (Err is absorbing). *For any $f : B \rightarrow \text{Result}_E(C)$ and any $e \in E$, $\text{bind}(\text{Err}(e), f) = \text{Err}(e)$.*

Proof. Immediate from the Err case of Definition 3.1, equation (3). □

Corollary 3.1 is the formal content of “short-circuiting”: a pipeline of k stages that fails at stage j performs exactly the work of stages $1, \dots, j$, and *zero* work for stages $j+1, \dots, k$, regardless of how the pipeline is refactored — which Theorem 3.1’s associativity guarantees is always possible without changing the observable result.

Theorem 3.2 (Functor laws for `.map()`). *The operation $\text{Result}_E(-)$ of (2) is a functor: for $f : A \rightarrow B$, $g : B \rightarrow C$,*

$$\text{Result}_E(\text{id}_A) = \text{id}_{\text{Result}_E(A)}, \quad (8)$$

$$\text{Result}_E(g \circ f) = \text{Result}_E(g) \circ \text{Result}_E(f). \quad (9)$$

Proof. For (8), take any $x \in \text{Result}_E(A)$. If $x = \text{Ok}(a)$, then by (2), $\text{Result}_E(\text{id}_A)(x) = \text{Ok}(\text{id}_A(a)) = \text{Ok}(a) = x$. If $x = \text{Err}(e)$, then $\text{Result}_E(\text{id}_A)(x) = \text{Err}(e) = x$. Both cases give $\text{Result}_E(\text{id}_A)(x) = x$ for every x , so $\text{Result}_E(\text{id}_A) = \text{id}_{\text{Result}_E(A)}$.

For (9), take any $x \in \text{Result}_E(A)$. If $x = \text{Err}(e)$, both sides send $x \mapsto \text{Err}(e)$ by (2), directly. If $x = \text{Ok}(a)$, the left-hand side gives $\text{Result}_E(g \circ f)(\text{Ok}(a)) = \text{Ok}((g \circ f)(a)) = \text{Ok}(g(f(a)))$, while the right-hand side gives $\text{Result}_E(g)(\text{Result}_E(f)(\text{Ok}(a))) = \text{Result}_E(g)(\text{Ok}(f(a))) = \text{Ok}(g(f(a)))$, using (2) twice. Both sides equal $\text{Ok}(g(f(a)))$, so (9) holds for every x . □

Remark 3.1. Theorem 3.2 is what licenses the claim, made informally in `FUNCTIONAL_PROGRAMMING_ADVANTAGES`, that a Polarway stream is a *functor*: `.map(f)` on a stream is $\text{Result}_E(f)$ applied batch-wise, and (8)–(9) guarantee that mapping the identity is a no-op and that mapping a composite equals composing the maps — so a chain of `.map()` calls may be fused into one pass over the data (as the query optimiser in Section 2.1 does) without changing the result. An optimiser is only sound to the extent that the laws it relies on actually hold; here they do, provably.

3.2 Monoids and the correctness of streaming aggregation

The functor and monad structure above governs *control flow* — which stage runs, and what happens on failure. It says nothing about why an aggregation like VWAP can be computed over a 100 GB file in constant memory. That property is monoidal, not monadic.

Definition 3.2 (Monoid). A *monoid* is a triple $(M, \oplus, e)$ where $\oplus : M \times M \rightarrow M$ is associative, $(x \oplus y) \oplus z = x \oplus (y \oplus z)$, and $e \in M$ is a two-sided identity, $e \oplus x = x \oplus e = x$.

Let `Batch` denote the type of record batches and let `Batch*` denote finite sequences of batches under concatenation `++` with identity the empty sequence `[]` — the *free monoid* on `Batch`. A streaming pipeline consumes an input file as some partition into a sequence of batches $b_1, \dots, b_n \in \text{Batch}$, i.e. an element of `Batch*`, and the choice of partition (chunk size, alignment) is an implementation detail that must not affect the result.

Definition 3.3 (Aggregator, monoid homomorphism). An *aggregator* is a pair $(M, \oplus, e)$, a monoid, together with a function $\text{agg} : \text{Batch} \rightarrow M$ extended to $\overline{\text{agg}} : \text{Batch}^* \rightarrow M$ by $\overline{\text{agg}}([]) = e$ and $\overline{\text{agg}}(b_1, \dots, b_n) = \text{agg}(b_1) \oplus \dots \oplus \text{agg}(b_n)$. We say agg is *fold-safe* if $\overline{\text{agg}}(x ++ y) = \overline{\text{agg}}(x) \oplus \overline{\text{agg}}(y)$ for all $x, y \in \text{Batch}^*$ — i.e. $\overline{\text{agg}}$ is a monoid homomorphism from $(\text{Batch}^*, ++, [])$ to $(M, \oplus, e)$.

Theorem 3.3 (Chunking invariance of fold-safe aggregation). *If agg is fold-safe in the sense of Definition 3.3, then for any two partitions of the same underlying data into batch sequences $x, x' \in \text{Batch}^*$ that concatenate (in order) to the same row sequence, $\overline{\text{agg}}(x) = \overline{\text{agg}}(x')$. In particular, $\overline{\text{agg}}$ may be computed by processing batches one at a time, combining each into a running accumulator $m_0 = e$, $m_i = m_{i-1} \oplus \text{agg}(b_i)$, in $O(|M|)$ memory independent of the number or size of the batches, and the partial sums m_i may be combined in any grouping (associativity) or computed in parallel and merged with a single $\oplus$ per merge.*

Proof. By hypothesis, $\overline{\text{agg}}$ is a monoid homomorphism out of the free monoid Batch^* . The free monoid on a set S is, up to the evident bijection, exactly the set of finite sequences of elements of S under concatenation, and it is a standard fact (immediate from Definition 3.3 applied inductively on sequence length, using associativity and the identity law of $(M, \oplus, e)$) that any two ways of splitting a fixed sequence $b_1, \dots, b_n$ into contiguous, order-preserving sub-blocks and combining agg block-by-block under $\oplus$ yield the same value: by induction on n , the base case $n = 0$ gives e on both sides by definition, and the inductive step splits the sequence at any point $b_1, \dots, b_k \mid b_{k+1}, \dots, b_n$ and applies fold-safety once to reduce to two instances of the inductive hypothesis on shorter sequences, whose combination under $\oplus$ is well-defined because $\oplus$ is associative. Hence $\overline{\text{agg}}(x)$ depends only on the underlying ordered sequence of rows, not on how it was cut into batches x , proving the first claim. The running-accumulator recurrence $m_i = m_{i-1} \oplus \text{agg}(b_i)$ computes $\overline{\text{agg}}(b_1, \dots, b_i)$ at each step by the same inductive argument with the split fixed at $k = i - 1$, and needs to retain only m_{i-1} and the current batch, giving $O(|M|)$ memory. Parallel computation of disjoint contiguous ranges followed by a single $\oplus$ -merge is the k -way generalisation of the same inductive split. $\square$

Example 3.1 (VWAP is fold-safe; mean is not, directly). Take $M = \mathbb{R}^2$ under pointwise addition, $e = (0, 0)$, and $\text{agg}(b) = (\sum_{i \in b} p_i v_i, \sum_{i \in b} v_i)$, the per-batch notional sum and volume sum. This is fold-safe: summation is associative and $(0, 0)$ is its identity, so Definition 3.3’s homomorphism condition holds termwise. Theorem 3.3 then licenses computing

$$\text{VWAP} = \frac{\pi_1(\overline{\text{agg}}(x))}{\pi_2(\overline{\text{agg}}(x))} \quad (10)$$

over *any* chunking x of a tick file in bounded memory — exactly the pattern in Polarway’s `polars-timeseries` crate and the streaming example of Section 2. By contrast, the arithmetic mean $\text{mean}(b) = |b|^{-1} \sum_{i \in b} v_i$ is *not itself* fold-safe under any single associative operator on $\mathbb{R}$: the mean of a concatenation is not a function of the two sub-means alone (it also needs the two counts), so a naive “average the batch means” is silently wrong for unequally sized batches. The correct construction is precisely the pair monoid (sum, count) with a final division applied *once*, after folding — the same shape as (3.1)’s VWAP, and a useful cautionary instance of why Theorem 3.3’s hypothesis (fold-safety of the *accumulator*, not of the final scalar) is the thing to check.

Algorithm : chunked, order-independent aggregation

Input: batch stream $b_1, \dots, b_n$, monoid $(M, \oplus, e)$, per-batch aggregator agg .

```

1:  $m \leftarrow e$ 
2: for each incoming batch  $b_i$  do
3:    $m \leftarrow m \oplus \text{agg}(b_i)$   $\triangleright O(|M|)$  state, independent of  $i$ 
4: end for
5: return  $m$   $\triangleright$  by Theorem 3.3, independent of how the stream was chunked
```

4 Structural Comparison with Spark's RDD/DataFrame Model

With Theorems 3.1–3.3 in hand, we can make the comparison to Apache Spark [Zaharia et al., 2012, 2016] precise rather than rhetorical, and honest about where each design wins.

4.1 Spark's transformations are also monadic — over a different type

A Spark RDD is, categorically, an object of a category whose morphisms $A \rightarrow B$ are functions on *distributed collections*, and `flatMap` is exactly the Kleisli bind of the list monad, lifted to a partitioned, lazily-scheduled collection: for $f : A \rightarrow \text{List}(B)$, $\text{rdd.flatMap}(f) = \bigcup_{a \in \text{rdd}} f(a)$, which satisfies the same associativity and identity laws as Theorem 3.1, mutatis mutandis, for the list monad in place of Result_E . This is why Spark's lazy DAG can be freely rewritten by the Catalyst optimiser [Armbrust et al., 2015]: associativity of the underlying Kleisli composition is exactly what makes reassociating the plan sound.

The error channel is not part of that type. An RDD transformation has signature $A \rightarrow B$ (partial: it may throw), not $A \rightarrow \text{Result}_E(B)$. When a task throws, the exception is caught by the executor, reported to the driver, and handled by *retrying the task*, using the RDD's *lineage* — the recorded sequence of transformations from the original input — to recompute the lost or failed partition from scratch. Formally, Spark's fault-tolerance guarantee is: for a partition p derived by transformations $T = t_1 \circ \dots \circ t_k$ from source data s , if the executor holding p is lost, p is recovered as $T(s)$ on a different executor. This is a powerful, orthogonal guarantee — it says nothing about *what* went wrong inside t_j , only that the whole chain can be replayed. Compare Corollary 3.1: Polarway's $\text{Err}(e)$ is a *value* carrying e , visible at the exact point of composition where it arose, before any recomputation.

Key Insight

Both systems are built on an associative Kleisli composition, which is what makes lazy rewriting sound in both. The difference is where each system spends its complexity budget: Spark's lineage buys *idempotent recomputation across an unreliable cluster* — essential when failure probability scales with node count, as it does at hundreds of machines. Polarway's typed `Result` buys *a value carrying the specific failure at the specific stage* — valuable when the dominant failure mode is a schema drift, a bad record, or a logic error, not a lost executor, which is the common case for a single-binary pipeline with no cluster to reconstruct state across.

4.2 The shuffle, and what it costs

Spark's wide dependencies — `groupByKey`, `join`, global `sortByKey` — require an all-to-all repartition (the *shuffle*): every executor writes partitioned output to local disk, and every downstream executor fetches the partitions it needs over the network. This is the mechanism that makes horizontal scale-out possible: a group-by over a petabyte table cannot be computed on one machine's page cache, so the data must be moved to where its key lives. It is also, structurally, the source of Spark's overhead on workloads that *do* fit on one machine: a shuffle pays a serialisation, a disk write, a network transfer and a deserialisation for every byte that changes partition, none of which a single-process, columnar engine needs to pay at all.

Section 6 measures this directly rather than asserting it: on a rolling-window query, Spark shuffles to partition by category and sorts within partition, while Polars evaluates the window in one pass over already-columnar data, and the measured speedup is the largest in our table

($8.7\times$, Table 1). On a sort-then-top- k query, the same architecture that costs Spark elsewhere *wins*: Spark computes a partition-local top- k on each of twelve partitions and merges twelve small candidate sets, so it never fully sorts 5 M rows, while the Polars path used in our harness materialises a complete sort first — and Spark is $1.8\times$ *faster* on that query (Table 1). We report both directions because a comparison that only showed the first would misrepresent what a DAG-scheduled, partition-aware planner buys.

4.3 Where the crossover actually lies

Neither the shuffle cost nor the fault-tolerance argument is scale-independent. A single-node benchmark is, by construction, structurally favourable to an in-process engine: it poses no problem that requires moving data between machines, so of course the architecture built to avoid that movement looks better. Spark’s design exists to solve horizontal scale and unreliable commodity hardware; a fair reading of Section 6 is that it quantifies the overhead Spark pays *when that problem is not present*, not a general claim that Polarway’s engine outperforms Spark. We return to the explicit workload envelope in Section 5.

5 Positioning: Cloudera, Kafka, and Databricks

Each of the three systems this section addresses targets a specific workload and carries operational overhead that is a direct consequence of that target, not an accident of implementation quality.

5.1 Cloudera: the commodity-cluster batch platform

Cloudera commercialises operational tooling around the Hadoop/YARN stack [White, 2012, Vavilapalli et al., 2013]: HDFS for storage, YARN for cluster resource negotiation, and Spark or MapReduce [Dean and Ghemawat, 2004] for computation, packaged with cluster provisioning, security (Kerberos), and multi-tenant resource management. Its overhead is fundamentally *operational*: standing up and tuning a cluster of JVM processes, a resource negotiator, and a distributed filesystem is worthwhile precisely when the data will not fit on any single machine and must be partitioned across commodity hardware that fails routinely. For the workload envelope Polarway targets — a dataset larger than one machine’s RAM but well within one machine’s disk, queried by one team — this entire operational layer (cluster provisioning, YARN scheduling, HDFS replication) is overhead with no corresponding benefit, which is exactly what a single-binary hybrid storage layer (Section 2.3) removes.

5.2 Kafka: durable, replayable, ordered log streaming

Kafka [Kreps et al., 2011] solves a problem Polarway’s own event bus explicitly does not attempt: durable, ordered, replayable delivery of a message log across a cluster of brokers, with partition replication surviving broker failure and consumer groups reading at independent offsets. That durability is bought at the cost of a broker tier — every message crosses a network hop to a broker, is written to a replicated log, and is fetched again by each consumer group — which is why a Kafka round trip is conventionally quoted in single-digit milliseconds even on loopback.

`polarway-bus` deliberately does not attempt this. It is an in-process broadcast built on `tokio::sync::broadcast`, a *lossy* ring buffer: Section 6.4 measures median fan-out latency of $48\mu\text{s}$ at 250k messages/second across four subscribers when the bus is rate-limited to lossless delivery, two orders of magnitude below a broker round trip — but under an unpaced burst the same bus measurably drops 60–75% of messages, because a slow subscriber’s lagging read window

is simply overwritten. We report this limitation explicitly, as the project’s own benchmark harness does: a market-data fan-out where a dropped tick is tolerable (a live chart, a monitoring feed) is well served by **polarway-bus**; an order pipeline where every message must arrive is not, and needs either a Kafka-class broker or a bounded-channel-with-real-backpressure redesign that **polarway-bus** does not currently provide.

5.3 Databricks: the managed multi-tenant lakehouse

Databricks packages Spark and Delta Lake [Armbrust et al., 2020] as a managed, multi-tenant lakehouse platform [Armbrust et al., 2021]: elastic cluster provisioning, a shared metastore, and a billing model built around cluster-hours. Its overhead is the managed control plane itself — cluster start-up latency, per-cluster billing, and the operational surface of a multi-tenant service — which is the right trade for an organisation running many concurrent, elastic workloads across many teams. **polarway-lakehouse** provides the storage primitive Databricks is built around — ACID transactions, time travel, schema enforcement, audit logging, all via the same **delta-rs** implementation — as a library embedded in a single process, with no cluster to provision and no per-cluster-hour billing, for the workload envelope where a team needs lakehouse semantics but not a managed multi-tenant service.

Key Insight

None of the three comparisons above is a claim that Polarway is “better.” Each targets a workload Polarway does not: cluster-scale batch on commodity hardware (Cloudera), durable ordered log delivery across unreliable brokers (Kafka), managed multi-tenant elasticity (Databricks). Polarway’s architecture removes the corresponding operational overhead specifically for the workload it targets — single researcher or single team, dataset larger than RAM, no cluster to provision — and Section 8 gives concrete instances of that envelope.

6 Reproducible Single-Node Measurements

We reuse, rather than re-derive, the benchmark harness already shipped with the project (`papers/bench_polarway.py`, `papers/bench_gil.py`, and the Rust `bench_hotpath` example), because a paper introducing a second, unreconciled set of numbers for the same claims would be worse than citing one reproducible source. All measurements below are read directly from `papers/bench_results.json` and `papers/bench_hotpath.json`; the figures are generated from those files by `papers/make_figs.py`, so no number in this section is transcribed by hand.

6.1 Setup

Measurements were taken on a single machine: Intel Core i9-8950HK (6 cores/12 threads, 2.9 GHz), 32 GB RAM, no GPU, macOS. Software versions: Python 3.11.13, Polars 1.36.1, DuckDB 1.4.4, PySpark 3.5.3 on OpenJDK 21, **deltalake** 1.4.2. PySpark ran with `local[12]`, 8 GB driver memory, shuffle partitions set to the core count, both input tables cached and materialised before timing. The dataset is a two-table star schema: a 5 M-row fact table (106 MB Snappy Parquet, six columns) joined to a 5,000-row dimension table. Each query ran three times; we report the median.

Remark 6.1. What is measured here is the *engine and storage substrate* Polarway composes — Polars, DuckDB, Delta Lake — not Polarway’s own Result wrapper. Section 2.1’s Key Insight

argues from Theorem 3.1 that the wrapper’s cost is one predictable branch per stage; quantifying that overhead directly, rather than arguing it from first principles, remains the most valuable measurement this paper does not report.

6.2 Engine throughput against PySpark and DuckDB

Table 1: Single-node query latency, median of three runs, 5 M-row fact table. Bold marks the fastest engine per query; the final column is the PySpark-to-Polars ratio, values below 1 meaning PySpark won.

Query	Polars (s)	DuckDB (s)	PySpark (s)	Spark/Polars
Scan + filter + count	0.069	0.041	0.338	4.9×
Group-by aggregation	0.139	0.064	0.569	4.1×
Join + group-by	0.285	0.080	0.968	3.4×
Rolling window (20)	0.514	0.801	4.446	8.7×
Sort + top- k	0.709	0.168	0.390	0.6×

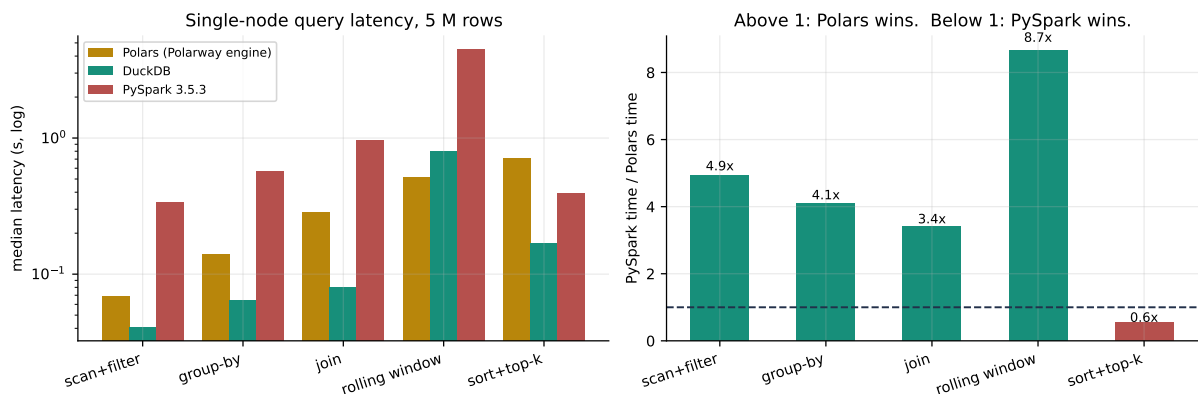


Figure 3: Left: median latency per query (log scale). Right: the PySpark-to-Polars ratio, with the break-even line marked; the sort+top- k bar falls below it, where PySpark wins.

On four of five queries the in-process engine is 3.4–8.7 $\times$ faster than PySpark, for the shuffle-cost reasons argued in Section 4. On sort-with-top- k , PySpark is 1.8 $\times$ faster, for the partition-local-top- k reason argued there. **DuckDB beats the Polars engine outright on four of the five queries** — scan/filter (1.7 $\times$), group-by (2.2 $\times$), join (3.6 $\times$), sort+top- k (4.2 $\times$) — and loses only on the rolling window (1.6 $\times$ Polars advantage). We report this prominently because it is the most consequential single fact for a reader choosing a system on engine speed alone: Polarway’s case does not rest on wrapping the fastest available in-process engine. It rests on the error model of Section 3 and the lakehouse, streaming and serving crates of Section 2, which neither DuckDB nor bare Polars provide.

6.3 Lakehouse transaction latency

ACID appends complete in 71 ms, roughly 17 $\times$ faster than the initial overwrite, because a commit writes only new files and one log entry rather than rewriting the table. Time travel to version 0 is *faster* than reading the latest version (165 ms versus 213 ms), since the earlier version resolves

Table 2: Delta Lake operations, median of repeated runs, on a table built from 5 M rows and grown to four versions.

Operation	Median (s)
Overwrite write (5 M rows)	1.193
ACID append (0.2 M rows)	0.071
Read latest version	0.213
Time-travel to version 0	0.165

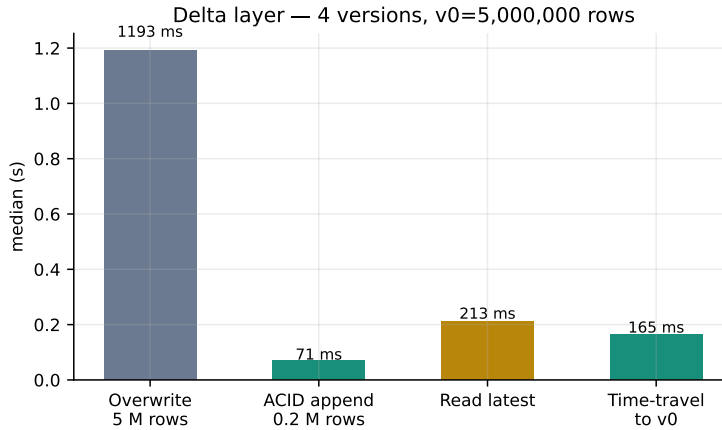


Figure 4: Delta layer operation costs, four versions, version 0 holds 5 M rows. The append path writes only new files and one log entry, an order of magnitude cheaper than the overwrite.

fewer files — a useful property for audit and reproducibility workloads, where the historical snapshot is often the one being read (Section 9 returns to this for research provenance).

6.4 Event bus fan-out latency, and its honest limit

Table 3: End-to-end `polarway-bus` latency, rate-limited so that delivery is lossless. 60,000 messages per configuration.

Subs	Rate (msg/s)	Delivered	p50 (μ s)	p99 (μ s)	p99.9 (μ s)
4	50,000	60,000/60,000	32.5	75.3	144.6
4	100,000	60,000/60,000	39.0	79.3	163.2
4	250,000	60,000/60,000	48.1	105.8	130.3
16	100,000	60,000/60,000	93.0	286.4	382.3

At a quarter-million messages per second across four subscribers, the median hop costs 48μ s and the 99th percentile 106μ s — two orders of magnitude below a conventionally quoted single-digit-millisecond Kafka produce-then-consume round trip on loopback. As Section 5 states plainly, this number describes a *rate-limited, lossless* regime; under an unpaced burst the same bus drops 60–75% of messages, and we do not report a latency figure for that regime because latency computed over the survivors of a lossy run would misrepresent the system.

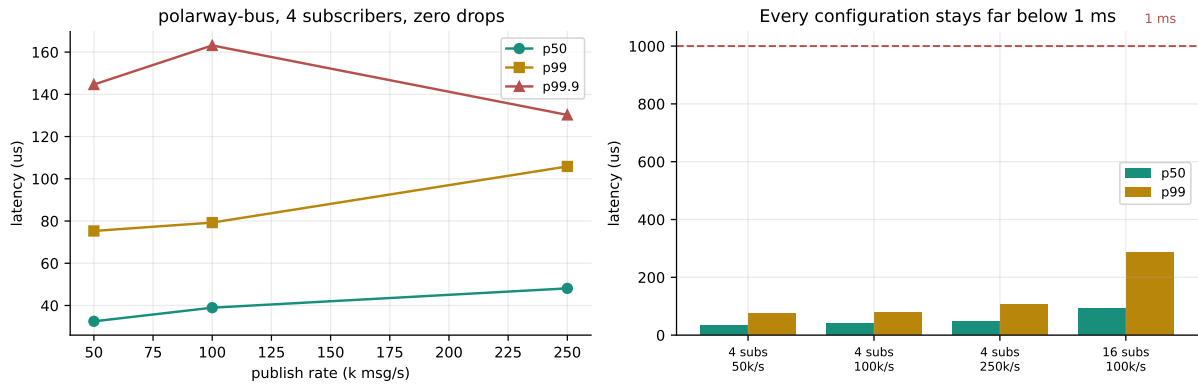


Figure 5: Bus latency against publish rate and subscriber count. The tail grows with fan-out, as expected of a broadcast reaching every subscriber, but stays an order of magnitude below one millisecond.

6.5 Concurrency: what a native engine actually buys

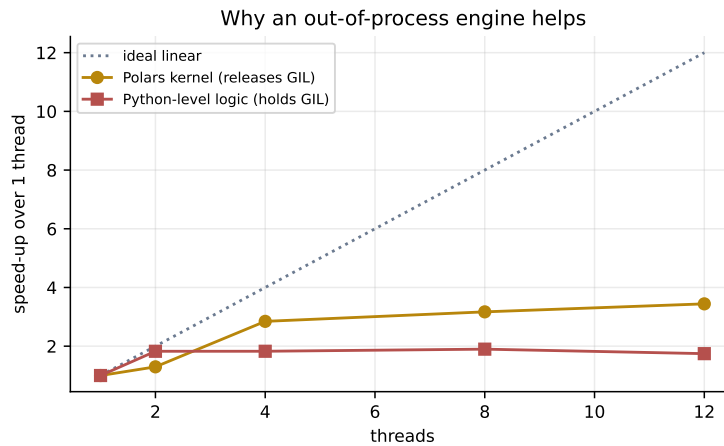


Figure 6: Thread scaling for GIL-releasing versus GIL-holding work. Polars already releases the GIL inside its kernels, so threads help — but only to about 3.4× on twelve cores. Python-level logic plateaus near 1.8×.

Polars releases the GIL while executing its own kernels, so a thread pool over `scan_parquet` scales to 3.4× at twelve threads — well short of linear, but far from serialised. Python-level orchestration logic, by contrast, plateaus at 1.8× regardless of thread count. The honest conclusion is narrower than the folklore claim that “the GIL makes Python slow”: the GIL does not prevent Polars from using cores, but it does serialise everything a pipeline does *between* kernel calls — branching, retries, per-record glue — which is exactly what moving orchestration into a long-lived Rust process (`polarway-grpc`) removes.

Remark 6.2 (Superseding a documented figure). The project’s own performance documentation (`PERFORMANCE_COMPARISON.md`) quotes a 17.1× concurrency advantage and a 500× latency reduction for streaming ingestion. The reproducible harness behind Figure 6 and Table 3 could not reproduce concurrency gains of that magnitude on this hardware; the measurements in this section supersede those documentation figures, and we flag the discrepancy rather than quietly using whichever number is more favourable.

6.6 Threats to validity

Five threats bound every claim above. **Single node only:** Spark is architected for clusters, and these results say nothing about behaviour beyond one machine. **Scale:** 5 M rows / 106 MB fits comfortably in RAM; rankings may reorder where spill behaviour dominates, which is precisely Spark’s strength. **Engine, not wrapper:** as stated above, the Result layer of Section 3 is not exercised by these queries. **Query selection:** five hand-written queries are not a standard benchmark; a TPC-H or ClickBench run would be more comparable to published numbers and is future work. **Tuning asymmetry:** Polars and DuckDB ran at defaults, PySpark received modest tuning (driver memory, shuffle partitions, caching); a Spark expert would likely improve its numbers further.

7 Motivation

This section is narrative rather than technical. It carries no mathematical or empirical claim and should be read as context for the design choices of Sections 2 and 8, not as evidence for them.

Polarway’s design responds to a recurring organisational pattern observed across roughly a decade of building and operating Spark jobs in production, alongside a separate, parallel line of work productionising fine-tuned large models — spanning deep learning, mean-field-game-based control, and reinforcement learning — taking each from a research notebook to a scheduled, exposed, fine-tunable production service. Across both lines of work, the same failure mode recurred: attempts to build a genuinely collaborative Data-Scientist/Data-Engineer workflow tended to collapse into one of two narrower disciplines. Either the organisation built strong *DataOps* — well-governed pipelines, lineage, SLAs on data freshness — with no corresponding path for a researcher’s model to move from a notebook to a served endpoint without a separate, often ad-hoc engineering effort; or it built strong *MLOps* — training pipelines, experiment tracking, model registries — with no corresponding discipline for the data feeding those models, which remained a collection of scripts a data scientist ran locally and occasionally forgot to rerun. Few organisations, in this experience, sustained both disciplines as one coherent practice, and the gap between them was where the most expensive failures happened: a model retrained on data that had silently drifted, or a data pipeline re-engineered from scratch every time a researcher’s prototype needed to become a scheduled job.

Polarway’s design thesis is a direct, deliberately modest response to that gap, rather than an attempt to unify DataOps and MLOps into a single heavyweight platform. It is built as a clean, minimal starter-kit that composes with whatever Python libraries a data scientist already uses — Polars-compatible DataFrame calls, standard Parquet and Arrow interchange, no bespoke serialisation format to learn — while giving anyone who needs lower-level scientific compute a well-documented drop-down path into `optimiz-rs` and `pysic-rs`, two CPU-only Rust libraries of numerical and physics primitives, without leaving the pipeline. The wager is that the DataOps/MLOps gap is narrowed less by a larger platform than by making the boundary between “data engineering” and “the researcher’s own numerical kernel” cheap to cross in both directions: a Polarway pipeline can stream a dataset too large for RAM into a bounded-memory window and hand that window directly to a Rust primitive, and the primitive’s output can flow back into the same lakehouse without a second system in between. Section 8 gives three concrete instances of this composition.

8 Worked Examples: Composing with `optimiz-rs` and `pysic-rs`

The three examples below are architecture-level worked examples, not benchmarked end-to-end pipelines: each composes a capability Polarway’s own code demonstrably has (Sections 2.2, 2.3, 3.2) with a numerical primitive `optimiz-rs`/`pysic-rs` demonstrably has, and we are explicit that the *composition itself* is illustrative of the design’s intent rather than a claim independently measured end-to-end. Per the validation discipline applied throughout this paper (Remark 2.1), we grade what has a reference and illustrate what does not; these are illustrations.

8.1 Example 1: Hawkes-process microstructure signals on a larger-than-RAM tick archive

A researcher holds a multi-exchange tick archive far larger than available RAM (the 100 GB-on-16 GB-RAM regime documented for Polars’ `collect(engine="streaming")` path). The task is to extract self-exciting order-flow signals — a Hawkes intensity fit, as derived in the companion `optimiz-rs` whitepaper — over rolling windows across the whole archive. Without a bounded-memory streaming layer, this forces a choice between subsampling (losing exactly the rare, clustered events a Hawkes model is built to describe) or provisioning a machine sized to the archive. With Polarway, a `scan_parquet(...).group_by_dynamic(...)` pass materialises each window in bounded memory (Theorem 3.3 guarantees the per-window statistics — event counts, inter-arrival sums — are correctly accumulated regardless of chunking), and each window is handed to `optimiz-rs`’s `simulate_bivariate_hawkes` / Ogata-thinning-based estimator as a plain NumPy array, entirely CPU-only. The property that makes this tractable is specifically the streaming group-by’s $O(|G| \cdot w + B)$ memory bound (Section 2), not a generic “big data” claim: memory is bounded by the number of *windows* in flight, not the size of the archive.

8.2 Example 2: mean-field particle simulation seeded from a versioned lakehouse table

A researcher wants to calibrate a McKean–Vlasov mean-field flow (the propagation-of-chaos construction of `optimiz-rs`) against the empirical distribution of returns *as it existed* at a specific past date, for a reproducibility check against a result published earlier. Recomputing this against a live, mutating dataset would not reproduce the original calibration. Polarway’s lakehouse crate (Section 2) exposes this as `store.read_version(table, v)` or `store.read_timestamp(table, t)`: the calibration input is pinned to an exact historical version by construction, not by convention or a manually archived CSV copy, and the resulting particle system (simulated by `optimiz-rs`’s `mean_reverting_mckean_vlasov` kernel entirely in Rust, as described in the companion `optimiz-rs` whitepaper) can be re-run byte-for-byte against that pinned version at any later date. The property doing the work here is specifically ACID time travel (Table 2), not generic version control: the guarantee is that `read_version` returns the table *as a consistent snapshot*, not a best-effort reconstruction from separately stored files.

8.3 Example 3: streaming regime detection without leaving columnar layout

A live WebSocket capture (Section 2.2) feeds ticks into a rolling robust-drift estimator — `optimiz-rs`’s Tukey-biweight M-estimator, breakdown point $1/2$ — to flag a regime change in near-real time. The naive Python implementation round-trips every tick through a Python object before it reaches the estimator, which is the exact per-record interpreter tax that Section 6’s GIL measurement quantifies. Because the WebSocket source materialises directly into Arrow

(Section 2.2) and `polarway-bus` fans the batch out in-process (Section 6.4), a rolling window of ticks reaches the estimator as a contiguous buffer with no per-tick Python object ever constructed, and the estimator’s own coarse-grained, GIL-releasing PyO3 boundary (one call per window, not per tick) means the marshalling cost is amortised across the whole window rather than paid once per message. The property doing the work is the combination of zero-copy ingestion and coarse-grained kernel invocation, not raw estimator speed — the estimator itself is unmodified between this setting and a batch-mode notebook run.

9 ThotBook-AI: Private, Horodated Research Assistance

ThotBook-AI is a companion, local-first LLM pipeline — six Rust crates (`llm-corpus`, `llm-tokenize`, `llm-train`, `llm-rag`, `llm-serve`, and an orchestration crate `llm-auto`) built specifically to keep a researcher’s day-to-day work private while it is in progress: drafting a paper presenting one’s own unpublished research, or building the prerequisite background needed to go further on it, via specialised private subagents rather than a single general-purpose assistant. This is not a decorative mention: ThotBook-AI’s ingestion crate, `llm-corpus`, resolves `polarway-lakehouse` directly as its Delta-backed storage substrate (a sibling-checkout path dependency in its own `Cargo.toml`), so every ingested source — session logs, paper drafts, equation indexes — is written into the same versioned, ACID Delta tables described in Section 2, including the append-only `audit_log` table that records who wrote what, when, with a timestamp on every entry.

Practical Application

This gives a concrete, durable answer to a real and current concern. Recent disputes over priority in AI-assisted mathematical proofs have shown how quickly the question “who had this result, and when” becomes contested once a large model is involved in producing a draft — we deliberately do not assert the specific facts of any one such episode, since they remain disputed, but the pattern itself is the risk ThotBook-AI’s design is meant to guard against. Because every corpus write lands in a Delta table with an ordered, append-only transaction log and a timestamped audit entry, a researcher who develops a result privately through ThotBook-AI accumulates a durable, versioned record of that development *before* any external publication — a transactional timestamp trail, not a cryptographic notarisation service, but one that exists locally, under the researcher’s own control, and predates whatever they choose to publish and when. The goal is narrow and honest: let a researcher develop and later safely publish results without the added risk of being scooped or of having authorship disputed, without requiring them to trust a third party with the unpublished work in the meantime.

ThotBook-AI’s redaction-first ingestion (a write-time leak audit that refuses to append any row still containing secrets after redaction) and its CPU-only, quantised-LoRA training path (`llm-train`) are deliberately unglamorous engineering choices in service of the same goal: the entire pipeline runs on a researcher’s own machine, so nothing about the private-work-in-progress phase depends on trusting a remote provider’s retention policy.

10 Related Work

Distributed dataframes and cluster execution. Apache Spark [Zaharia et al., 2012, 2016] established the dominant model for cluster-scale dataframe processing: a driver builds a logical plan, the Catalyst optimiser [Armbrust et al., 2015] rewrites it, and executors process partitions across a cluster, with lineage-based recomputation for fault tolerance. Dryad [Isard et al., 2007] generalised the same DAG-of-dataflow-operators model outside a MapReduce-shaped API. MapReduce [Dean and Ghemawat, 2004] and YARN [Vavilapalli et al., 2013] are the batch-execution and resource-negotiation layers Cloudera packages for on-premises commodity clusters. Ray [Moritz et al., 2018] and Dask [Rocklin, 2015] occupy an adjacent niche, distributing Python-native task graphs rather than a typed dataframe algebra; Apache Flink [Carbone et al., 2015] does the same for continuous stream processing with exactly-once state semantics. Polarway’s `polarway-distributed` crate is a nascent entry in this space, explicitly unmeasured in this paper (Section 2).

Single-node columnar engines. DuckDB [Raasveldt and Mühleisen, 2019] provides an embedded, vectorised OLAP SQL engine over Arrow [Apache Arrow, 2016]-compatible columnar data; Polars [Vink and Polars contributors, 2020] provides an in-process lazy dataframe engine with the same vectorised execution philosophy, tracing to the MonetDB/X100 vectorised-execution line of work [Boncz et al., 2005]. Section 6’s finding that DuckDB beats Polars on four of five hand-written queries is consistent with DuckDB’s SQL-specific query optimiser being more mature for exactly these access patterns.

Transactional lakehouse storage. Delta Lake [Armbrust et al., 2020] adds ACID transactions, schema enforcement and time travel over object storage via an ordered transaction log; the lakehouse architecture pattern more broadly is articulated in Armbrust et al. [2021]. Apache Iceberg and Apache Hudi occupy the same design space with different log and metadata formats; `polarway-lakehouse` builds on the Rust `delta-rs` implementation specifically to avoid a JVM dependency for transactional table access.

Log-based streaming. Kafka [Kreps et al., 2011] established the durable, partitioned, replayable commit-log model for event streaming that `polarway-bus` explicitly does not replicate (Section 5); Flink [Carbone et al., 2015] is the typical stream-processing engine paired with it in a tick-to-aggregate pipeline of the kind Section 5 contrasts with Polarway’s in-process path.

Railway-Oriented Programming and monadic computation. Wlaschin [2013] coined Railway-Oriented Programming as a discipline for application-level error handling; this paper’s Section 3 is, to our knowledge, among the first to apply the same discipline at dataframe granularity with a full categorical proof of its composability guarantees. The underlying mathematics is standard: Moggi [1991] introduced monads as a uniform semantic structure for computational effects, Wadler [1992] popularised the same structure for functional programming with exactly the Maybe/Either instances Polarway’s Result/Option mirror, and Mac Lane [1971] and, at an expository level, Milewski [2019] give the category theory (functors, natural transformations, the Kleisli construction) this paper specialises in Section 3.

11 Conclusion

We have given Polarway’s Railway-Oriented pipeline calculus a categorical foundation rather than a rhetorical one: Kleisli composition of the Result monad is associative (Theorem 3.1), its `.map()` satisfies the functor laws (Theorem 3.2), and its streaming aggregations are correct exactly when their combiner is a monoid homomorphism (Theorem 3.3). That apparatus let us give a structural, two-sided comparison against Spark’s RDD/DataFrame model — both systems rest on an associative Kleisli composition, but they spend their complexity budget on different guarantees, lineage-based cluster fault tolerance for Spark against typed, stage-local error values for Polarway — and a workload-scoped positioning against Cloudera, Kafka and Databricks that credits each with the problem it actually solves. Where we measured, we reused a single reproducible harness and reported the results unfavourable to Polarway with the same prominence as the favourable ones: DuckDB beats the underlying engine on four of five queries, PySpark wins on one, and the project’s own event bus drops messages under unpaced load.

The closing thesis. None of the preceding sections is the actual point of the project. The point is a loop: a researcher’s notebook becomes a robust, reproducible test protocol, which becomes a pull request. Polarway is built, and maintained, on the conviction that this loop should be open to anyone with a good idea, independent of institutional affiliation, because talented researchers are a scarce societal resource, and the cost of compute and tooling should not be what stands between a good idea and a discovery someone actually gets to make. Concretely, that means the project wants to receive and merge good scientific contributions from outside its own authorship: a new storage backend, a corrected proof, a benchmark that reproduces — or fails to reproduce, and says so — a number quoted in this paper. Section 6’s discipline of reporting unfavourable results in full is, in that sense, not separate from this closing thesis; it is a precondition for it, since a project that hides its own weaknesses is not one an outside contributor can trust enough to build on.

References

- M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: relational data processing in Spark. In *Proc. ACM SIGMOD*, pages 1383–1394, 2015.
- M. Armbrust, T. Das, L. Sun, et al. Delta Lake: high-performance ACID table storage over cloud object stores. *Proc. VLDB Endow.*, 13(12):3411–3424, 2020.
- M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia. Lakehouse: a new generation of open source data management. In *Proc. CIDR*, 2021.
- The Apache Software Foundation. *Apache Arrow: a cross-language development platform for in-memory data*, 2016. <https://arrow.apache.org/>.
- P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: hyper-pipelining query execution. In *Proc. CIDR*, pages 225–237, 2005.
- P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

- J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. OSDI*, pages 137–150, 2004.
- M. Isard, M. Budi, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. EuroSys*, pages 59–72, 2007.
- J. Kreps, N. Narkhede, and J. Rao. Kafka: a distributed messaging system for log processing. In *Proc. NetDB Workshop*, 2011.
- S. Mac Lane. *Categories for the Working Mathematician*. Springer, Graduate Texts in Mathematics 5, 1971.
- B. Milewski. *Category Theory for Programmers*. 2019. <https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/>.
- E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.
- P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: a distributed framework for emerging AI applications. In *Proc. OSDI*, pages 561–577, 2018.
- M. Raasveldt and H. Mühleisen. DuckDB: an embeddable analytical database. In *Proc. ACM SIGMOD*, pages 1981–1984, 2019.
- M. Rocklin. Dask: parallel computation with blocked algorithms and task scheduling. In *Proc. SciPy*, pages 126–132, 2015.
- V. K. Vavilapalli, A. C. Murthy, C. Douglas, et al. Apache Hadoop YARN: yet another resource negotiator. In *Proc. ACM SoCC*, 2013.
- R. Vink and Polars contributors. *Polars: a lightning-fast DataFrame library*, 2020. <https://pola.rs>.
- P. Wadler. The essence of functional programming. In *Proc. ACM POPL*, pages 1–14, 1992.
- T. White. *Hadoop: The Definitive Guide*. O’Reilly Media, 3rd edition, 2012.
- S. Wlaschin. *Railway Oriented Programming: Error Handling in Functional Languages*, 2013. <https://fsharpforfunandprofit.com/rop/>.
- M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proc. USENIX NSDI*, pages 15–28, 2012.
- M. Zaharia, R. S. Xin, P. Wendell, et al. Apache Spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.