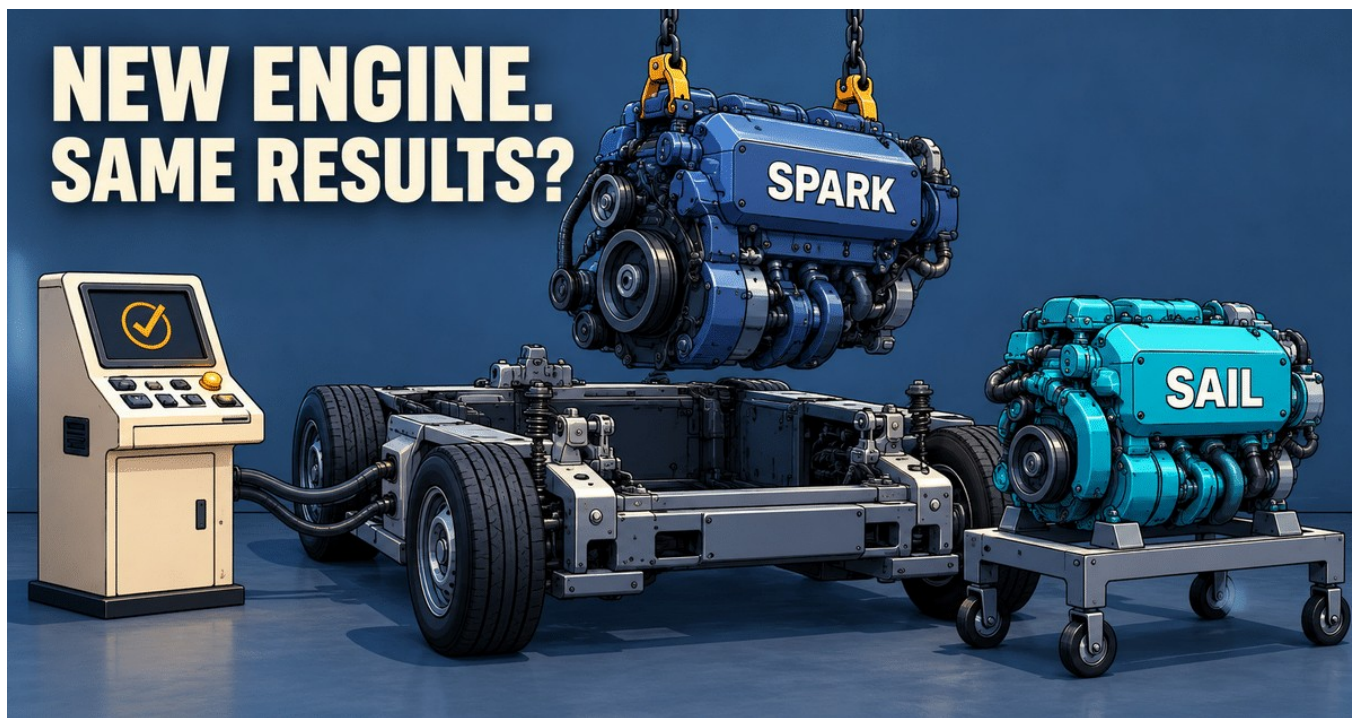


Sail vs Spark: Architecture, Benchmarks, Dask and Numba

Julien Laurenceau | Published 2026-06-14 | Updated 2026-10-22

<https://pepitedata.com/sail-apache-spark-rust-practitioner-deep-dive/>

PepiteData | LinkedIn: Julien Laurenceau | X: @Julien_Lau



A familiar client does not guarantee identical results after an engine change. Validate the complete workload.

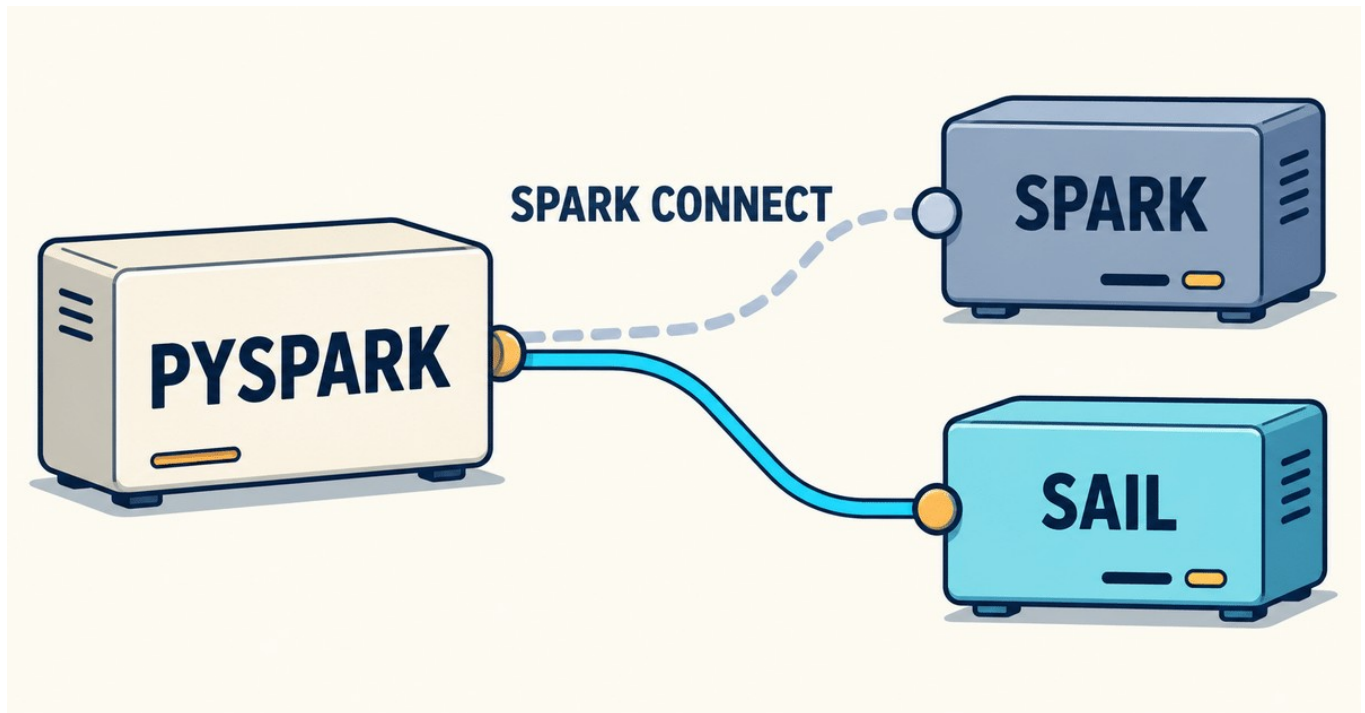
Sail can run Spark SQL and DataFrame workloads on a Rust execution engine through Spark Connect. That makes it worth evaluating when execution overhead matters. It does not make every Spark application compatible, or turn a published benchmark into a forecast for your bill.

I would start with a simpler question: which part of the workload needs to change? Sail changes the engine. Dask distributes Python work. Numba compiles suitable numerical functions. Choosing between them requires identifying the expensive operation before choosing a language or a framework.

What does Sail replace in Spark?

Sail provides a Spark Connect server backed by a Rust execution engine. A compatible PySpark client sends plans to Sail instead of a Spark server. Sail uses DataFusion and Arrow within its execution stack. The supported client API can remain familiar while the server, physical plans and operational behavior change.

In local mode, execution runs in one multithreaded process. In a Sail cluster, the driver schedules work across workers; Arrow Flight carries result and shuffle data. These are architectural choices to evaluate, not evidence that every workload will benefit. See the [Sail architecture documentation](#).



Spark Connect separates the client from the execution server. The two endpoints are alternatives, not simultaneous execution paths.

There is an important correction to the usual Python-versus-JVM story: [Spark Connect already uses gRPC rather than Py4J](#), including when the server is Apache Spark. Sail changes server execution; it does not invent that client separation. Python UDFs also remain Python code, so a native engine does not imply that every part of the application becomes Rust.

Is Sail a Spark plugin?

Sail is an alternative Spark Connect server, whereas Apache DataFusion Comet is a plugin inside Spark. Comet accelerates supported physical operators and can fall back to Spark for unsupported execution. The distinction matters because a replacement engine and an acceleration plugin retain different parts of the existing Spark application and operating environment.

The [Comet project documentation](#) describes its native execution and fallback model. I would compare that approach with Sail when retaining Spark integrations is important. A shared DataFusion foundation does not make their compatibility, deployment or failure behavior identical.

What do Sail's benchmarks actually establish?

Sail's published benchmark establishes a result for a specified test, not a general production speedup. Checked on October 22, 2026, the derived TPC-H report compares Sail 0.7.0 with Spark 4.2.0 across 22 queries on one AWS r8g.4xlarge instance. These are publisher measurements, not a PepiteData replication.

Reported metric	Spark 4.2.0	Sail 0.7.0
Sum of query execution times	534.78 seconds	52.81 seconds
Peak memory	72 GB	26 GB
Shuffle writes	115 GB	0 GB

Publisher-reported derived TPC-H results. The shuffle result applies to this test, not to all Sail workloads.

The [methodology](#) specifies scale factor 100, Parquet, 16 vCPUs and 128 GB of memory. Changed data generation and file sizing prevent direct comparison with older reports.

The publisher's 98% cost estimate combines runtime with a smaller-memory pricing assumption, not a full customer invoice. The [latest release checked was Sail 0.7.2](#), distinct from the benchmarked version.

Which compatibility checks come before a migration?

Sail compatibility needs to be checked against the application's actual APIs, data semantics and integrations. Spark SQL or DataFrame support does not imply support for RDDs, Java UDFs or every Spark configuration. A connection change can be short while the validation remains substantial, especially for writes, recovery and downstream consumers.

The [Sail migration guide](#) lists RDDs as unsupported and Structured Streaming as partially supported. Its scanner can help inventory code, but static checks do not execute the application or prove equivalent results. Spark Hadoop filesystem settings are another migration boundary: do not assume an existing storage configuration carries over unchanged.

The [UDF support documentation](#) includes Python UDFs and UDTFs, with support depending on the API used. Java and Scala UDFs are unsupported. A workload dominated by Python function execution needs its own measurements; native SQL execution alone does not establish its speed.

Validation area	What I would compare	Acceptance evidence
Results	Nulls, decimals, timestamps, joins and ordering where required	Matched outputs using defined tolerances and edge cases
Integrations	UDFs, storage authentication, catalogs and downstream readers	Representative end-to-end runs with required permissions
Operations	Writes, retries, worker loss, observability and rollback	Isolated output paths, tested recovery and a retained baseline

Suggested acceptance checks, not a claim that every integration is supported.

Where do Dask and Numba fit?

Dask and Numba address different Python performance problems. Dask coordinates partitioned or task-based computation across available workers. Numba compiles supported numerical functions for faster local execution. Neither is a drop-in Spark SQL engine, and neither should be selected merely because a pipeline contains Python. The expensive operation determines which approach is relevant.

Dask: partition Python work when distribution helps

[Dask DataFrame](#) operates on partitioned pandas objects, with lazy computation that can run on one machine or a cluster. It is a candidate for a pandas-oriented workflow that exceeds comfortable single-process capacity. Moving a PySpark program to Dask is a migration of APIs and execution assumptions, not a server endpoint change.

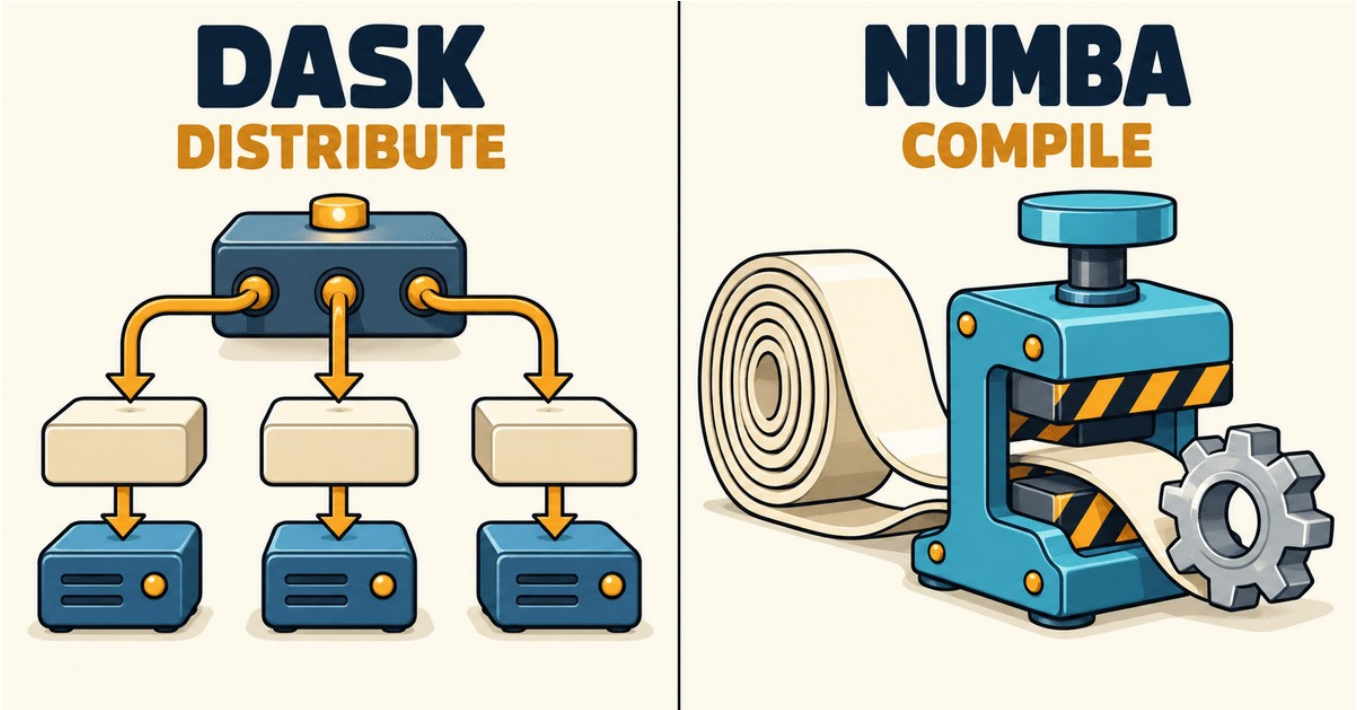
Partition size, shuffle and task overhead still matter. If pandas already handles the data comfortably, I would first test whether distributing the work adds anything. Dask's own [best practices](#) recommend improving algorithms and local computation before introducing more parallelism.

Numba: compile the numerical function

[Numba's JIT compiler](#) is useful for supported numerical code, particularly loops over NumPy arrays. It is not a compiler for an arbitrary pandas or Spark application. I would isolate the function, check supported

types and operations, and compare both its results and its timing with the existing implementation.

Report compilation and steady-state execution separately. A warm repeated numerical kernel and a short process paying compilation cost answer different questions. Numba does not provide a distributed scheduler, a SQL optimizer or storage governance simply because the compiled function is faster.



Dask distributes work; Numba compiles suitable functions. They address different costs and can sometimes be combined.

Candidate	Problem to investigate	Change boundary	Proof required
Sail	Supported SQL/DataFrame execution is costly	Spark Connect server and engine	Semantic parity, integrations and complete job cost
Spark with Comet	Supported Spark operators may benefit from native execution	Spark execution plugin	Operator coverage, fallback and end-to-end benefit
Dask	Python work needs partitioning or scheduling	Python computation graph and APIs	Partition behavior, memory, shuffle and scheduling overhead
Numba	A supported numerical function dominates CPU time	Function implementation and types	Correctness, compilation cost and repeated execution time

A selection guide, not a performance ranking. Profile before choosing a candidate.

Dask and Numba can also be evaluated together. For example, Dask could schedule independent array partitions while a Numba-compiled function processes each partition. That design only helps if useful computation outweighs transfer, scheduling and compilation costs. Check worker thread counts as well: two layers requesting parallel execution can compete for the same CPU capacity.

How would I benchmark the actual pipeline?

A useful engine evaluation measures correct end-to-end work under representative operating conditions. Include input reads, planning, Python functions, shuffle, writes and failure recovery where those affect the service. Record versions and configuration, separate cold and warm runs, and compare cost per accepted result rather than treating query execution time as the entire bill.

1. **Capture a baseline.** Record input snapshot, output requirements, hardware, software versions and elapsed time. Identify where CPU, memory, I/O or coordination dominates.
2. **Choose the smallest relevant experiment.** Test a supported Sail job, a Comet-enabled plan, a Dask partitioning approach or a compiled function according to the measured bottleneck.
3. **Validate correctness before speed.** Use representative edge cases and explicit numerical tolerances. Write to new locations; do not overwrite the baseline outputs.
4. **Measure the complete run.** Include startup, compilation, transfers and output commits when they recur in production. Repeat runs and retain variability rather than publishing the best attempt.
5. **Test operation and reversal.** Exercise relevant failures, estimate infrastructure and engineering cost, and retain a working rollback path before changing production routing.

Engine changes also interact with [file formats and physical layout](#). A new execution engine cannot be assumed to fix excessive reads or unsuitable partitioning. For the broader method, use my [performance and cost engineering guide](#). My [Spark smoke tests on Kubernetes](#) examine a different benchmark question: data access and distributed system behavior.

FAQ: Sail, Dask and Numba

Sail, Dask and Numba should be evaluated against a specific execution problem and a defined correctness requirement. API familiarity does not establish compatibility, and a faster component does not establish a cheaper pipeline. The practical questions concern supported operations, migration scope, complete elapsed time and the cost of operating the resulting system.

Can Sail replace every Spark application?

No. Evaluate Sail against the APIs and integrations actually used. RDDs and Java or Scala UDFs are unsupported in the documentation checked for this revision, and Structured Streaming support is partial. A successful SQL example does not validate an entire production application.

Is Dask a drop-in replacement for PySpark?

No. Dask provides its own APIs and scheduling model. A pandas-oriented distributed workflow may suit Dask, but an existing PySpark application needs an explicit migration assessment covering computation, memory, data movement and integrations.

Does Numba accelerate an entire data platform?

No. Numba compiles supported functions rather than replacing the whole platform. A faster numerical function helps the pipeline only to the extent that the function contributes to total runtime. Include compilation and surrounding data conversion in the measurement.

Should I budget for Sail's advertised savings?

Not without a representative evaluation. The published cost estimate combines benchmark runtime with an infrastructure-sizing assumption. Use your own accepted outputs, operating constraints, resource prices and migration effort to build the decision case.

Next step: test the expensive operation

Choose an optimization only after establishing which operation consumes the time or money you need to recover. A supported SQL job, a distributed Python graph and a numerical function require different tests. I would use a bounded comparison with agreed acceptance criteria before committing to a

platform change or a migration budget.

If you need help designing that comparison, my [Spark tuning and benchmark service](#) focuses on identifying bottlenecks and testing changes against the workload. No customer data or customer benchmark results are used in this article.

Updated October 22, 2026: refreshed the versioned Sail benchmark, corrected Spark Connect and compatibility claims, and added Dask and Numba selection criteria. Sources are linked at the claims they support. This independent technical assessment implies no affiliation with or endorsement by the projects discussed.

PDF snapshot of the public article. The online version is authoritative; links remain clickable.