LLNL-CFPRES-2022872

Lawrence Livermore National Laboratory

HORATIO: Bridging Management and Analysis of Traces at Scale

N Dryden, H Hariharan Devarajan, H Gunawi, R Sinurat, W Nixon

Aug 2026

38th International Conference on Scalable Scientific Data Management

San Diego, CA, United States of America

2026-08-12 through 2026-08-13

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

HORATIO

Bridging Management and Analysis of Traces at Scale

Highly Optimized Raw-trace Analysis, Trace Indexing, and Organization

Ray A. O. Sinurat · William Nixon · Haryadi S. Gunawi · Nikoli Dryden · Hariharan Devarajan

The University of Chicago · Lawrence Livermore National Laboratory





SSDBM '26 · San Diego, CA · August 11–13, 2026

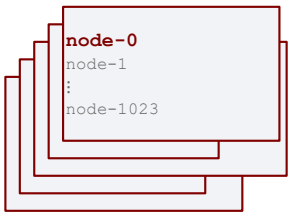
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

LLNL-CFPRES-2022872

Simple analysis, hard at scale

Your **thousand-node run** came back slower than it should have, so you traced it.

That leaves you with **thousands of files**: hundreds of GB to TBs uncompressed traces, and wildly uneven in size.

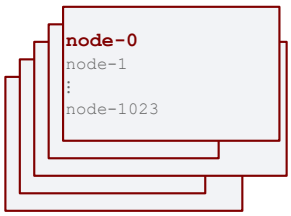


sizes vary widely

Simple analysis, hard at scale

Your **thousand-node run** came back slower than it should have, so you traced it.

That leaves you with **thousands of files**: hundreds of GB to TBs uncompressed traces, and wildly uneven in size.



sizes vary widely

The **first epoch** ran slow. Where did its time go?

One question. It should take minutes. But.. it doesn't.

Open one up

With DFTracer, each rank writes its own file: newline-delimited JSON, one event per line, gzip'd.

```
{"name":"write", "ph":"X", "ts":16993…, "dur":42, "pid":1000,
   "args":{"epoch":0, "hostname":"node-0", "step":7, "fhash":…}}
```

Open one up

With DFTracer, each rank writes its own file: newline-delimited JSON, one event per line, gzip'd.

```
{"name":"write", "ph":"X", "ts":16993…, "dur":42, "pid":1000,
   "args":{"epoch":0, "hostname":"node-0", "step":7, "fhash":…}}
```

fixed fields `name, ts, dur, pid`
every event has them

args `epoch, hostname, …`
free-form, no schema, and what you query on

Open one up

With DFTracer, each rank writes its own file: newline-delimited JSON, one event per line, gzip'd.

```
{"name":"write", "ph":"X", "ts":16993…, "dur":42, "pid":1000,
   "args":{"epoch":0, "hostname":"node-0", "step":7, "fhash":…}}
```

fixed fields `name, ts, dur, pid`
every event has them

args `epoch, hostname, …`
free-form, no schema, and what you query on

This is what DFTracer and Chrome-format profilers emit, and it's our use case.
The query we want, **`epoch == 0`**, lives right there in the free-form args.

What you actually ask of a trace

Rarely just one query. Usually some mix of these three.

Filter by time
pull a window: one step, one epoch, the first few minutes of a run.

Filter by criteria
match a predicate on the fields: epoch == 0, host == node-0, one file, one operation.

Aggregate
reduce the matches: total I/O time per file, event counts, p99 durations.

Both answers cost you

Leave it raw

Point Perfetto or a script at the files.

One copy, nothing to convert.

But every query scans **everything**.

Both answers cost you

Leave it raw

Point Perfetto or a script at the files.

One copy, nothing to convert.

But every query scans **everything**.

Convert to columnar

Load it into Parquet or DuckDB.

Fast queries, and they prune.

But you keep the raw too, so **two copies**, and the traces have no fixed schema.

Both answers cost you

Leave it raw

Point Perfetto or a script at the files.

One copy, nothing to convert.

But every query scans **everything**.

Convert to columnar

Load it into Parquet or DuckDB.

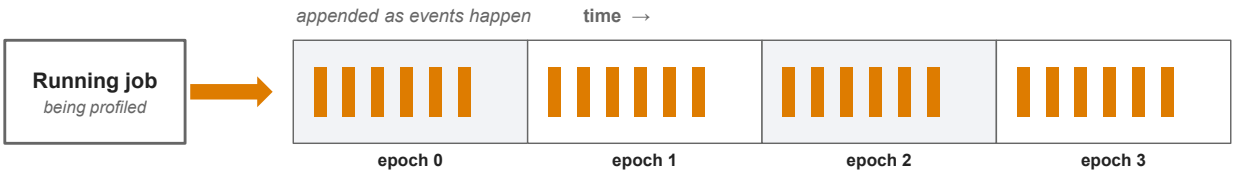
Fast queries, and they prune.

But you keep the raw too, so **two copies**, and the traces have no fixed schema.

Each buys one thing by giving up another.

Why not just write columnar?

The tracer runs inside the job it measures, so it writes the cheapest way it can.



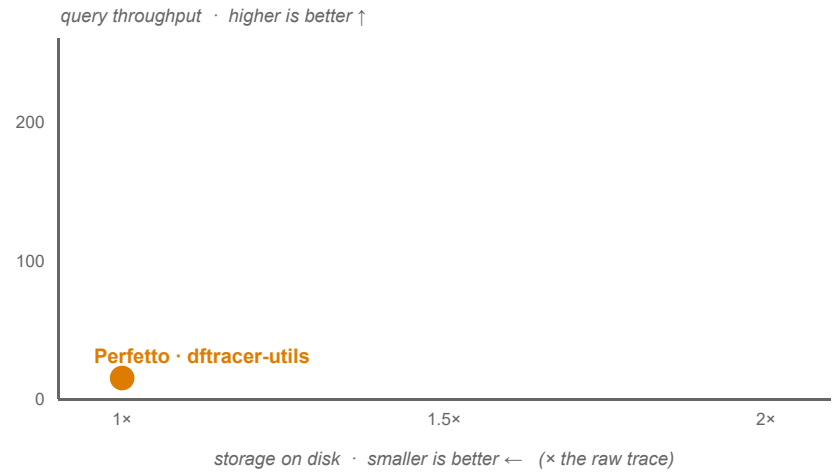
one line per event, row-oriented and already in time order

Columnar wants the opposite: buffer and transpose a batch into columns, and fix the schema in advance. But a trace's fields typically appear at runtime, from user annotations and whatever code paths get instrumented, so there is no fixed schema to give it.

The structure is already there, in time order. The raw format just doesn't expose it.

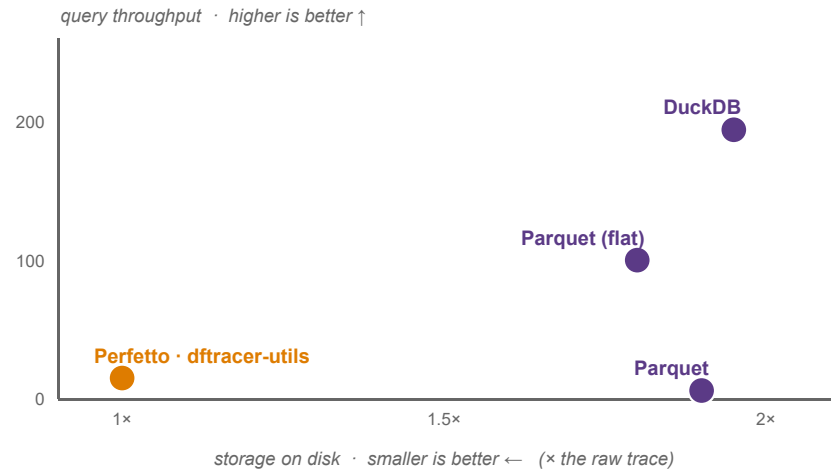
Fast or small, not both

Synthetic trace, ~1.2 B events (243 GB raw). One query: pull every event in an epoch.



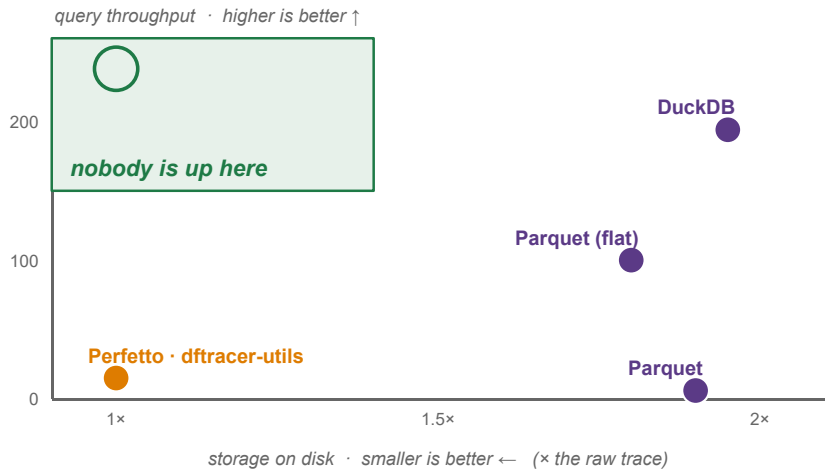
Fast or small, not both

Synthetic trace, ~1.2 B events (243 GB raw). One query: pull every event in an epoch.



Fast or small, not both

Synthetic trace, ~1.2 B events (243 GB raw). One query: pull every event in an epoch.



Fast *and* small (the top-left corner) is empty. That corner is what this talk is about.

Three limitations to overcome

Querying raw traces in place is hard for three reasons, and each one maps to a component.

No selective querying
raw traces have no entry points or predicate metadata, so even a tiny query scans everything.

answered by
the Indexer

Three limitations to overcome

Querying raw traces in place is hard for three reasons, and each one maps to a component.

No selective querying
raw traces have no entry points or predicate metadata, so even a tiny query scans everything.

answered by **the Indexer**

Analysis throughput
Pandas and Dask hit interpreter and scheduler walls at terabyte scale.

answered by **the Analyzer**

Three limitations to overcome

Querying raw traces in place is hard for three reasons, and each one maps to a component.

No selective querying
raw traces have no entry points or predicate metadata, so even a tiny query scans everything.

answered by **the Indexer**

Analysis throughput
Pandas and Dask hit interpreter and scheduler walls at terabyte scale.

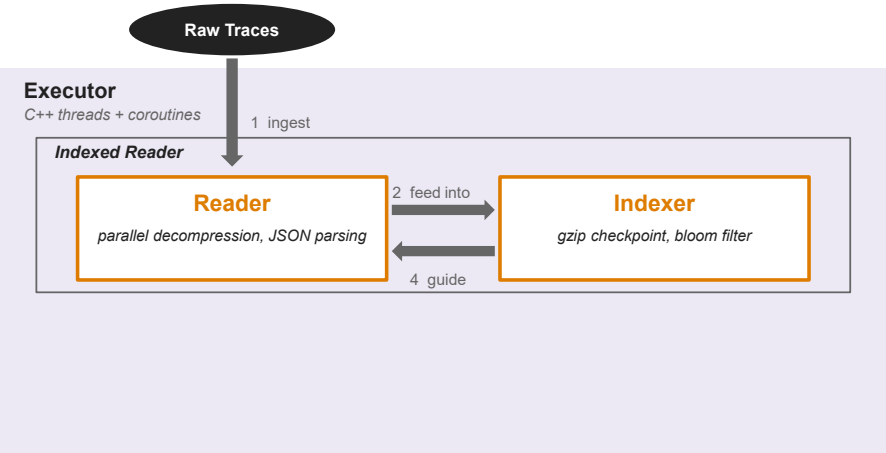
answered by **the Analyzer**

No physical clustering
matching events scatter across files, so a targeted query fans out over the whole dataset.

answered by **the Assembler**

HORATIO: five components, one raw copy

The whole design, one piece at a time. We deep-dive each one next.

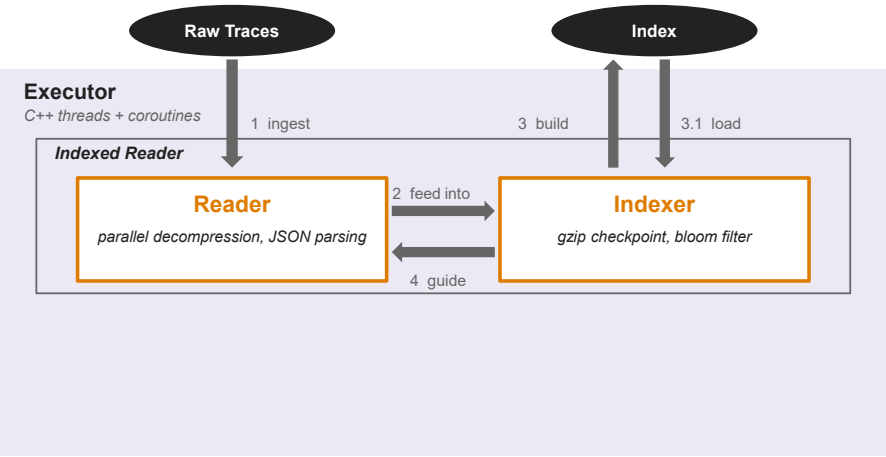


Reader
decode the raw files in parallel

Indexer
skip before you decode

HORATIO: five components, one raw copy

The whole design, one piece at a time. We deep-dive each one next.

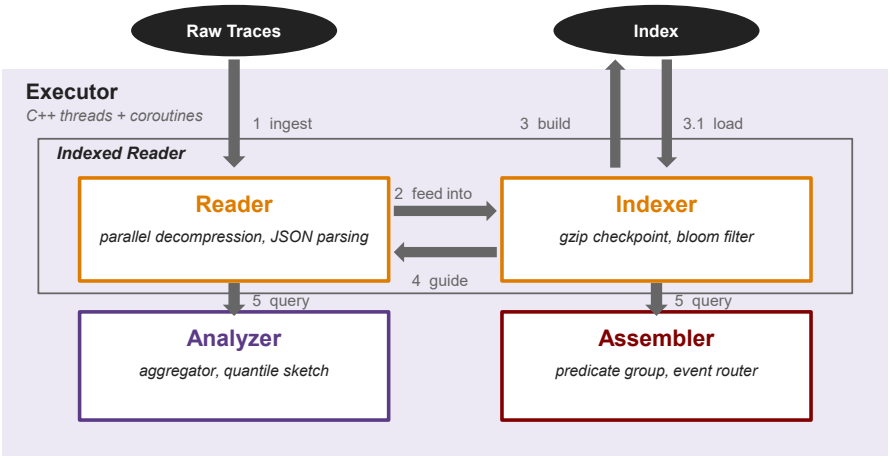


Reader
decode the raw files in parallel

Indexer
skip before you decode

HORATIO: five components, one raw copy

The whole design, one piece at a time. We deep-dive each one next.



Reader
decode the raw files in parallel

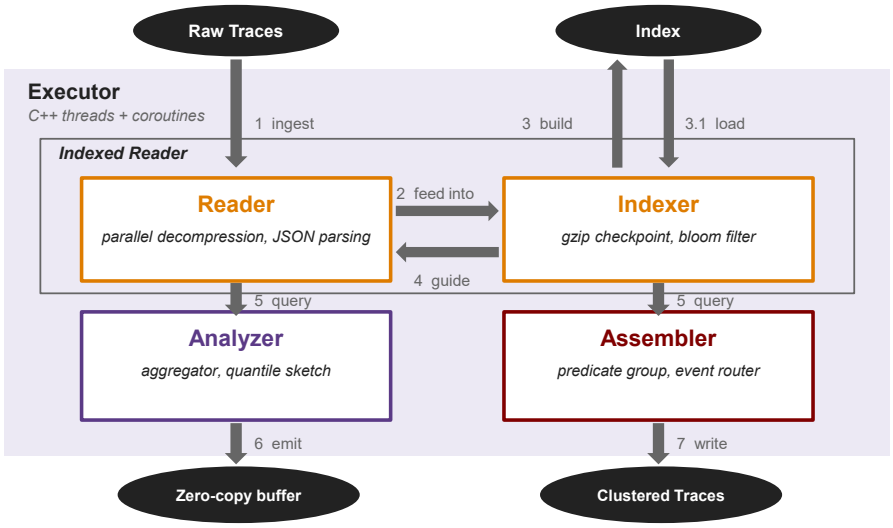
Indexer
skip before you decode

Analyzer
aggregate, off Python

Assembler
cluster, and stay raw

HORATIO: five components, one raw copy

The whole design, one piece at a time. We deep-dive each one next.



Reader
decode the raw files in parallel

Indexer
skip before you decode

Analyzer
aggregate, off Python

Assembler
cluster, and stay raw

Executor
runs it all, scales out

One raw copy, nothing converted.

Reader: parallel access to raw traces

The foundation. A single gzip file decodes sequentially, and that throttles everything downstream.

Inter-file parallelism

hundreds of per-rank files become independent decode units, dispatched to the Executor pool with no synchronization.

Intra-file parallelism

split a file on gzip-member boundaries; once an index exists, resume from zran checkpoints (bit offset and zlib state) for random-offset reads.

simdjson parsing

SIMD-accelerated, record by record, non-owning views so no per-event allocation, and arbitrary keys with no fixed schema.

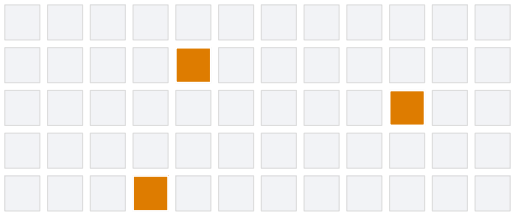
Indexer: skip before you decode

A raw trace is millions of small gzip chunks. A sidecar index lets a query rule most of them out.

Read it the plain way and you decode every one.
Under 20 M events/s.

Indexer: skip before you decode

A raw trace is millions of small gzip chunks. A sidecar index lets a query rule most of them out.



Read it the plain way and you decode every one.
Under 20 M events/s.

A per-trace sidecar index

bloom filters per chunk, from every field, even nested, no schema

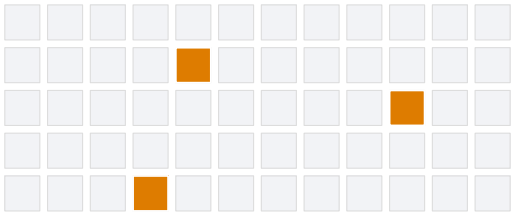
min/max + distribution statistics for range predicates

zran checkpoints for random access

all in RocksDB, not single-writer SQLite

Indexer: skip before you decode

A raw trace is millions of small gzip chunks. A sidecar index lets a query rule most of them out.



Read it the plain way and you decode every one.
Under 20 M events/s.

A per-trace sidecar index

- bloom filters per chunk, from every field, even nested, no schema
- min/max + distribution statistics for range predicates
- zran checkpoints for random access
- all in RocksDB, not single-writer SQLite

epoch 0 → only a handful of chunks survive. Everything else is skipped, untouched.

Analyzer: move event work off Python

Answers the second limitation: Pandas and Dask hit interpreter and scheduler walls at TB scale.

Native C++ event pipeline

the whole event path, parse, predicate, aggregate, runs in C++, not per-row Python dispatch on materialized dataframes.

Streaming aggregation

group by a composite key, slice into time windows (WisIO's main view), and merge DDSketch quantiles (1% error) with no barrier.

Zero-copy Arrow exchange

results become Arrow record batches Python reads directly over IPC, with no buffer copies or extra serialization.

Assembler: cluster events, keep the format

Answers the third limitation: write-order layouts scatter matching events across files.

Predicate-based clustering

consults the Indexer to find matching chunks, then routes each event into predicate-defined groups, e.g. I/O vs others.

Format-preserving, lossless

output stays gzip'd NDJSON, every line byte-exact, with per-group provenance to rebuild any original on demand, so no duplicate copy.

Lustre-aware chunking

writes tunable-size output chunks and aligns flush boundaries to the stripe size, avoiding small-file metadata cost.

Executor: the concurrency backbone

Shared by all four components. Thread-per-file hits OS limits at trace-scale parallelism.

Lazy C++20 coroutines

stackless, suspend-on-create, so each coroutine is a unit of scheduled work and the active set stays bounded.

Async channels

Reader feeds Indexer, Analyzer, or Assembler through lock-free channels, composing them into pipelines with automatic backpressure.

MPI for scale-out

each rank runs its own coroutine pool; small reductions via collectives, large index outputs via local RocksDB shards the coordinator bulk-ingests.

How we tested it

Corona @ LLNL: dual AMD Rome (96 threads), 256 GB, local SSD, Lustre at 240 GB/s.

Workload	Files	Compressed	Uncompressed
Synthetic	256	17 GB	~242 GB
Montage	60,649	15 GB	~275 GB
UNet3D	384	1.8 GB	~37 GB
Llama	20,993	525 MB	~4.3 GB
h5bench	512	41 GB	~2.2 TB

Five workloads

one synthetic, four real, across AI training and scientific computing, all profiled by DFTracer.

h5bench, 2.2 TB uncompressed, is the scaling stress test.

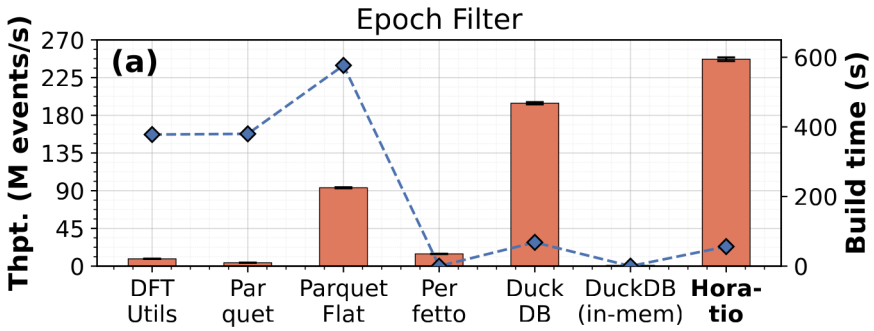
Baselines

Parquet, ParquetFlat, DuckDB, Perfetto, dftracer-utils, and the Dask-based DFAnalyzer.

ParquetFlat lifts every nested args field into its own column, the most query-favorable Parquet layout.

epoch_filter: finer skipping wins

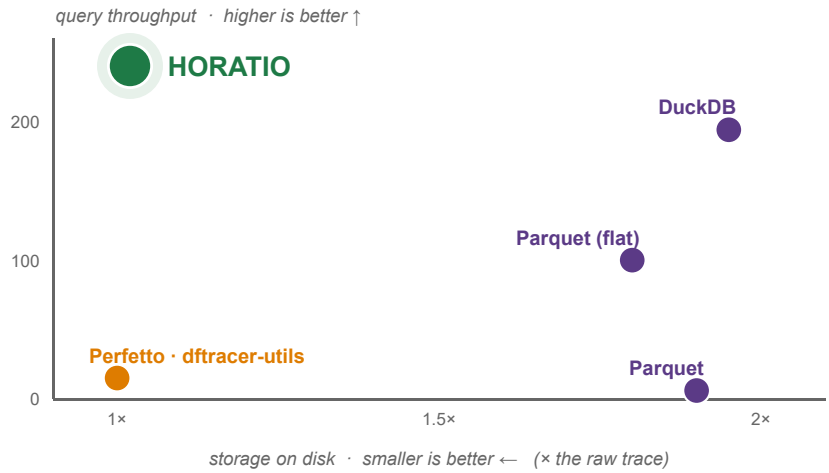
epoch == 0, pull one training epoch. Bars are query throughput (left axis); the dashed line is one-time build time (right axis).



247 M events/s

A time-ordered field, so chunk-level statistics prune far finer than row-group pruning: ahead of DuckDB (194 M) and ParquetFlat (94 M). Naive Parquet collapses to 4 M, with epoch buried in its args JSON.

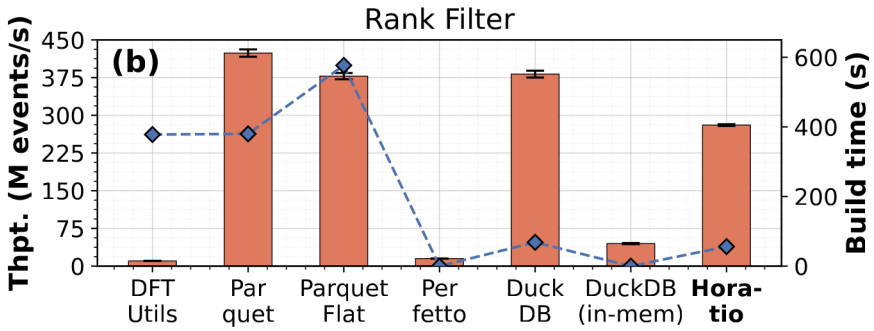
Into the corner



The epoch query: 247 M events/s, on a single raw copy at 1.01× storage.

rank_filter: the one place columnar wins

pid == 1000, isolate one rank. Each file is already one rank, so every format is a full single-file scan. Same axes as before.

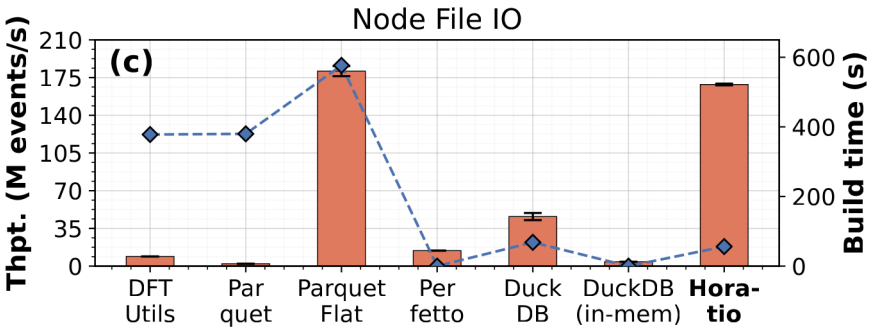


281 M events/s, 1.51× behind

Nothing to prune, so HORATIO still decompresses and parses each surviving chunk, while columnar reads a pre-decoded column. Parquet leads at 424 M.

node_file_io: a compound predicate, near tie

host == 'node-0' and file == …, per-file I/O on a node. A compound filter over nested args. Same axes.

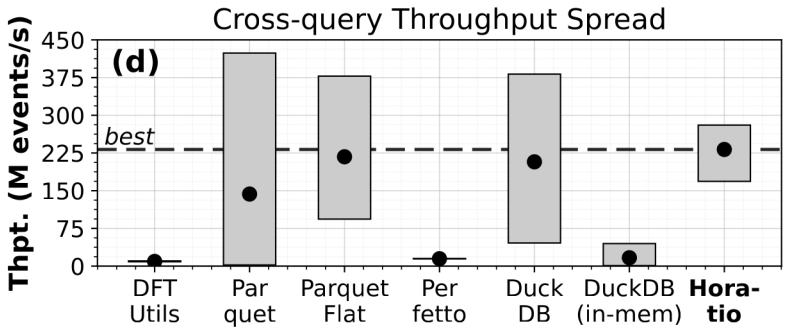


169 M events/s, within 1.07×

Per-chunk bloom filters skip nearly as finely as ParquetFlat's flattened columns (181 M). Surviving chunks still pay decompression and parsing, so the two land close.

Across all three: highest mean, tightest spread

Each format's min-to-max throughput over the three queries; the dot is the average, the dashed line marks the best average.

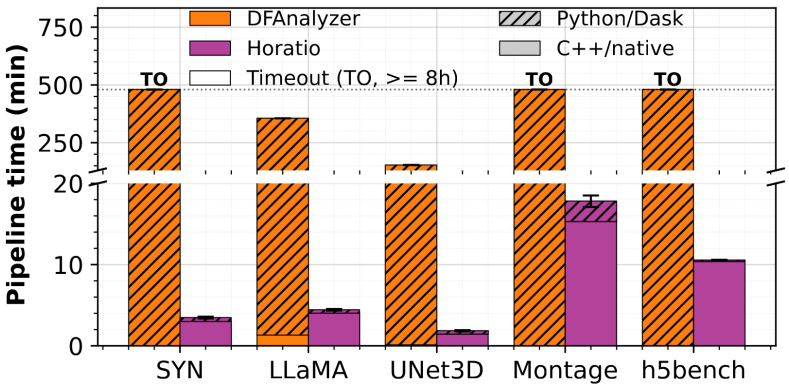


232 M events/s mean

We decode about 1.5× slower per byte but skip more aggressively, so finer pruning wins whenever it touches less data. The result is the tightest spread and the highest mean, on one raw copy at 1.01× storage.

Python was the bottleneck

A single query is one thing. Running a full analysis pipeline over the trace is another.

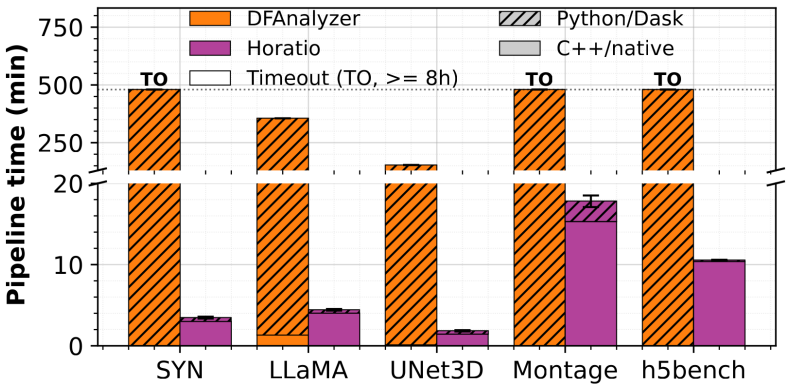


80–83× faster where both finish.

DFAnalyzer times out (≥ 8 h) on 3 of the 5 workloads.

Python was the bottleneck

A single query is one thing. Running a full analysis pipeline over the trace is another.



80–83× faster where both finish.

DFAnalyzer times out (≥ 8 h) on 3 of the 5 workloads.

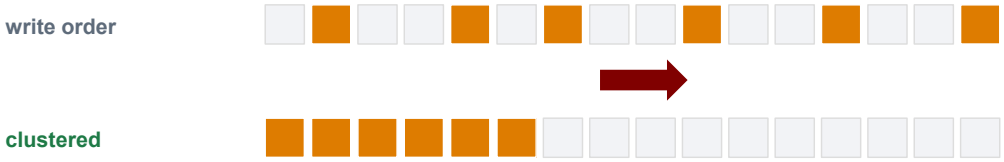
Why

the event math runs in native C++, streams, and hands back zero-copy Arrow.

Memory stays 0.4–3.8 GB, vs 9.5 GB+ (and OOM on 2.2 TB).

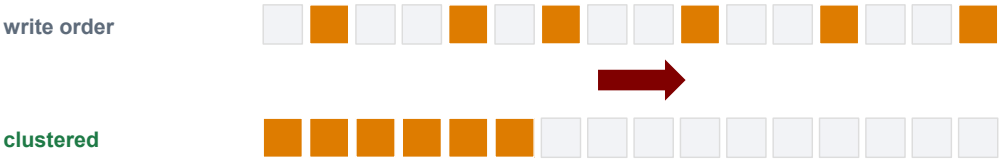
Cluster once, query cheap

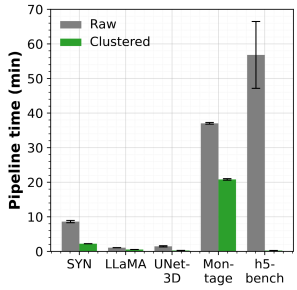
The Assembler reorganizes the raw layout once, grouping events by predicate.



Cluster once, query cheap

The Assembler reorganizes the raw layout once, grouping events by predicate.



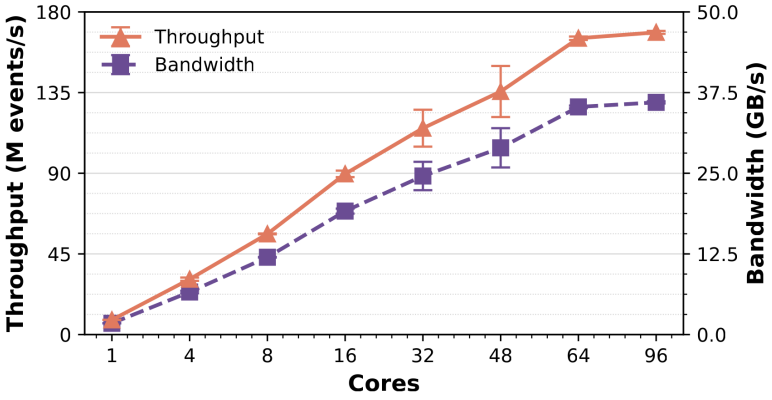


1.8× to 230× faster once it's clustered.

The layout grows only to 1.23–1.48× raw, and stays lossless and reversible.

First, inside one node

The selective node_file_io query on Synthetic, sweeping 1 to 96 threads. Left axis throughput, right axis bandwidth.



21× across 96 threads

8 M up to 169 M events/s, reaching ~36 GB/s.

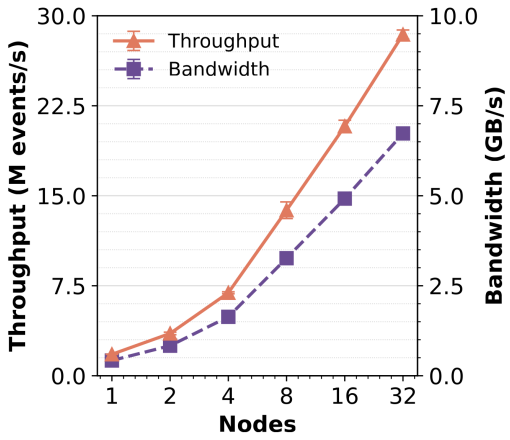
Why it tapers

steep through the 48 physical cores, where decode and parse are compute-bound.

past that, a selective query has too few surviving chunks to keep the extra hyperthreads busy.

And across nodes, at 2.2 TB?

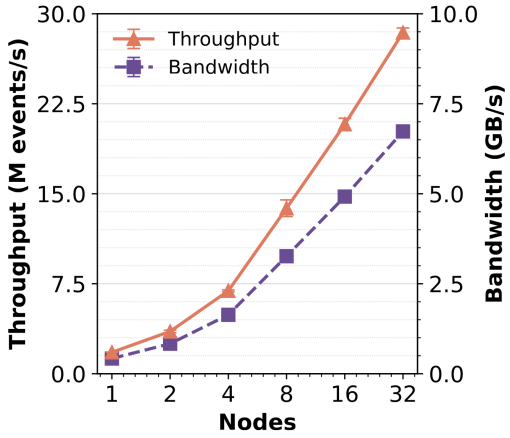
The full h5bench trace, 1 to 32 nodes, two ways to scale it out.



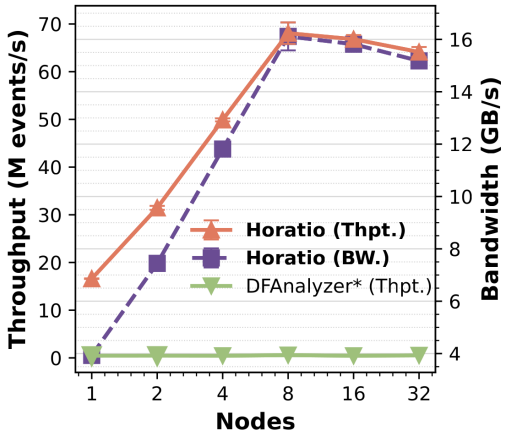
MPI aggregator: **16× at 32 nodes**

And across nodes, at 2.2 TB?

The full h5bench trace, 1 to 32 nodes, two ways to scale it out.



MPI aggregator: **16× at 32 nodes**



Dask backend 4×; DFAnalyzer hits a memory wall

Where this goes next

The index, the native analysis, and the clustering are not tied to the format.

Other profiler formats

Darshan and Recorder use binary records; the same in-place index applies once we add a sync-point scheme for their codec.

NUMA-aware scheduling

pin decode and index-probe work to the socket that owns the data, to lift the memory-latency ceiling we hit within a node.

An indexed backend for visualization

expose the Indexed Reader so a visualization UI skips chunks the same way our analysis path does.

A quick demo, on a laptop



Trace viewer: marketplace.visualstudio.com/items?itemName=rayandrew.dftracer-viewer

So: back to the question

The first epoch ran slow. Where did its time go?

A question you can now answer in seconds, on a single raw copy.

queried	analyzed	clustered
247 M events/s at 1.01× storage	80–83× faster than Dask DFAnalyzer	1.8–230× for every run after

Manage raw traces in place. No conversion, no second copy.

Thank you. Questions?

Ray A. O. Sinurat · rayandrew@uchicago.edu · github.com/rayandrew/horatio-artifact

THE UNIVERSITY OF CHICAGO

Lawrence Livermore National Laboratory