LLNL-CONF-2019625

Lawrence Livermore National Laboratory

Horatio: Bridging Management and Analysis of Traces at Scale

H Hariharan Devarajan

Sep 2026

38th International Conference on Scalable Scientific Data Management

San Diego, CA, United States of America

2026-08-11 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

Ray A. O. Sinurat
Department of Computer Science
University of Chicago
Chicago, Illinois, USA
rayandrew@uchicago.edu

William Nixon
Department of Computer Science
University of Chicago
Chicago, Illinois, USA
williamn@uchicago.edu

Haryadi S. Gunawi
Department of Computer Science
University of Chicago
Chicago, Illinois, USA
haryadi@cs.uchicago.edu

Nikoli Dryden
Lawrence Livermore National Laboratory
Livermore, California, USA
dryden1@llnl.gov

Hariharan Devarajan
Lawrence Livermore National Laboratory
Livermore, California, USA
hariharandev1@llnl.gov

Abstract

Modern scientific and deep learning workloads on HPC systems rely on profiling and tracing across multiple software and hardware layers, generating diagnostic traces that often reach terabyte scale. Existing approaches manage these traces along two axes: trace formats and analysis tooling. Practitioners often convert raw traces into queryable formats, but doing so nearly doubles storage when raw files are retained for compatibility and requires upfront schema discovery that profiling tools cannot guarantee. A cleaner path is to make raw traces efficient in place, but this requires overcoming three limitations: lack of selective querying, analysis throughput bottlenecks, and lack of physical clustering. To address these limitations jointly, we developed Horatio, a raw trace management framework that indexes, analyzes, and physically clusters raw traces directly. Three findings emerge from our work. First, Horatio stores a lightweight RocksDB-backed auxiliary index alongside the raw trace, including gzip checkpoints, per-chunk bloom filters, and chunk-level statistics, delivering selective queries up to 75× faster than naive Parquet at ~1.01× raw storage, with the highest cross-query mean throughput (232 M events/s) among state-of-the-art formats. Second, offloading event-level computation to a native C++ backend while keeping Dask for orchestration yields 80–83× speedup over the original Dask-based DFAnalyzer. Third, lossless trace clustering that preserves the same input format yields a further 1.8–5.5× end-to-end speedup across four AI and scientific workloads, and up to 230× on h5bench where the preset aligns tightly with the cluster boundary, all with original layouts reconstructible on demand. Across five AI and scientific workloads, Horatio completes pipelines that DFAnalyzer cannot finish within 8 hours. On a 2.2 TB uncompressed trace, Horatio's MPI-based mode scales to 16× at 32 nodes on the full event set and its Dask-based path peaks at 4× on a preset-filtered workload, both completing where DFAnalyzer hits OOM at every scale.

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor, or affiliate of the United States government. As such, the United States government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for government purposes only. Request permissions from owner/author(s).
SSDBM 2026, San Diego, CA, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2708-5/26/08
https://doi.org/10.1145/3828820.3828825

CCS Concepts

• **General and reference** → **Performance**; • **Computing methodologies** → *Concurrent algorithms*; • **Applied computing** → *Document management*; • **Information systems** → **Extraction, transformation and loading**.

Keywords

Horatio, parallel gzip, index, bloom filter, RocksDB, coroutine

ACM Reference Format:
Ray A. O. Sinurat, William Nixon, Haryadi S. Gunawi, Nikoli Dryden, and Hariharan Devarajan. 2026. HORATIO: Bridging Management and Analysis of Traces at Scale. In *The International Conference on Scalable Scientific Data Management 2026 (SSDBM 2026), August 11–13, 2026, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3828820.3828825

1 Introduction

Modern scientific and deep learning (DL) workloads span a deep, heterogeneous stack of storage, network, memory, and accelerators, and are inherently data-intensive, requiring large volumes of training data, checkpoints, and intermediate results. To optimize these workloads, practitioners rely on profiling and tracing across multiple layers of the stack. For instance, I/O profiling tools (e.g. DFTracer [40] and Darshan [60]) generate traces that capture storage-level behavior, while framework-level tools (e.g. PyTorch Profiler [28] and TensorFlow Profiler [14]) and hardware-level tools (e.g. CUPTI [16] and ROCM Profiler [30]) produce traces that expose computation and accelerator activity. Extensive analysis of these traces can reveal critical bottlenecks and yield substantial performance improvements across the pipeline.

Existing approaches address trace management primarily along two axes: trace formats and analysis tooling. On the format side, JSON-based formats have become particularly prevalent due to their human readability, broad tooling support, and compatibility with structured logging practices. Perfetto [20] supports both Protocol Buffers [22, 43] and Chrome JSON Trace Format [29], while I/O profilers such as Darshan [60] and Recorder [66] use custom binary formats, and DFTracer [40] adopts gzip-compressed Chrome JSON Trace Format, achieving over 90% size reduction. On the analysis side, practitioners rely on general-purpose data processing frameworks such as Pandas [17], DuckDB [56], Dask [5], and Polars [54]

for querying and aggregating traces, visualization tools such as Perfetto [20] and TensorBoard [62], and profiler-specific ecosystems such as DFAnalyzer [10] for DFTracer, PyDarshan [38] for Darshan, and WisIO [69], now shipped as the new DFAnalyzer in recent DFTracer. However, while these analysis tools scale to large datasets, they depend on the underlying trace format supporting efficient access patterns, which raw trace formats typically do not provide.

To bridge this gap, practitioners often convert raw traces into queryable formats such as Parquet [18], whose data-skipping primitives (row-group statistics, bloom filters, column projection) let readers prune irrelevant data without scanning the full file. Profiling tools typically emit row-major formats for their fast and append-only writes, since producing read-efficient columnar formats at trace time would incur significant write overhead and require upfront schema discovery. Furthermore, the original traces are rarely discarded since many raw formats remain directly consumable by existing downstream tools. Recorder, for instance, retains its native binary for visualization while exporting Parquet for downstream analysis frameworks such as WisIO [69]. DFTracer presents a related case where its gzip-compressed NDJSON is read directly by the Perfetto UI, text editors, and CLI tools (e.g. `zcat`, `jq`), yet querying it efficiently requires conversion to a separate format. This dual retention nearly doubles the storage footprint, reaching 32 GB for a single DFTracer workload (17 GB of gzip-compressed NDJSON plus 15 GB of Parquet, §3). The challenge is therefore to deliver columnar-grade query performance directly on the row-major raw format that profilers already produce, without forcing conversion.

Achieving this requires overcoming three fundamental limitations. First, lack of selective querying, as row-major raw trace formats provide no random-access entry points or predicate metadata, forcing even highly selective queries to decompress and scan entire files. Second, analysis throughput bottlenecks, where tools built on general-purpose frameworks (e.g. Pandas, Dask) hit performance walls at terabyte scale due to interpreted-language overhead and data serialization between framework layers. Third, lack of physical clustering with events matching a predicate scattered across many files, so queries fan out across far more files than necessary even when a within-file index is available. Dask and Polars can reorganize trace data only through shuffles or format conversion, both of which break raw-format compatibility and recreate the dual-retention problem. Together, these limitations underscore the need for a raw trace management framework that delivers selective querying, high-throughput analysis, and physical clustering of events, all while preserving the raw format that users rely on for downstream tooling.

To address these challenges, we developed Horatio[1], an efficient raw trace management framework. Horatio stands for **H**ighly **O**ptimized **R**aw-trace **A**nalysis, **T**race **I**ndexing, and **O**rganization. Horatio consists of five components: *Reader*, *Indexer*, *Analyzer*, *Assembler*, and *Executor* that together enable users to efficiently query, analyze, and scale trace processing directly on raw data without format conversion and with minimal storage overhead. To address the lack of selective querying, *Indexer* attaches per-chunk bloom filters, predicate statistics, and zran-style random-access checkpoints to the row-major raw trace through a RocksDB sidecar, delivering finer-grained data-skipping than columnar row-groups without any conversion. To address analysis throughput bottlenecks, *Analyzer* provides a native C++ analysis path with zero-copy data exchange to Python, reducing interpreted-language overhead while preserving flexibility for interactive exploration. To address the lack of physical clustering, *Assembler* adds a one-time lossless trace clustering into predicate-defined chunks, accelerating analyses on that predicate without keeping a duplicate of the original, since the original layout is reconstructible on demand. *Reader* provides parallel gzip decompression and JSON parsing that pairs with *Indexer* to form an Indexed Reader for downstream consumers. Finally, *Executor* is a coroutine-backed parallel execution backend that powers all four components, coordinating concurrent operations across fragmented trace files. Evaluation on five workloads spanning AI training and scientific computing demonstrates that Horatio delivers selective queries up to 75× faster than naive Parquet at ∼1.01× raw storage, leading the cross-query mean throughput (232 M events/s) with the lowest variability, up to 83× end-to-end pipeline speedup over WisIO's DFAnalyzer on completed workloads (and finishing three workloads where DFAnalyzer cannot complete within the budget), and a further 1.8–5.5× speedup from physical reorganization across four workloads (up to 230× on h5bench), all without format conversion. Horatio's coroutine-backed C++ runtime also shows strong scaling to 32 nodes on a 2.2 TB uncompressed trace when integrated with MPI and Python Dask pipeline. The contributions of this work are as follows:

(1) We design *Indexer*, a lightweight index layer that achieves the highest cross-query mean throughput (232 M events/s) with under 1% storage overhead.
(2) We build *Analyzer*, a native analysis pipeline with zero-copy Python exchange that delivers up to 83× end-to-end speedup over the original Dask-based pipeline.
(3) We develop *Assembler*, predicate-based trace clustering that yields 1.8–5.5× speedup across four workloads and up to 230× on terabyte-scale traces.

2 Background

2.1 Trace Format

Despite their diversity in scope, traces produced across the I/O, framework, and hardware layers share a common shape: each event carries a name, a timestamp, a duration, a process or thread identifier, and an arbitrary metadata payload, though granularity varies widely from per-operation I/O logs to pipeline summaries to hardware counter samples collected at microsecond intervals. A widely adopted serialization for these events is the Chrome JSON Trace Format [29] encoded as newline-delimited JSON (NDJSON), commonly gzip-compressed for storage efficiency and emitted as one file per rank or worker to avoid synchronization at trace time. This choice aligns with broader structured-logging conventions in observability and monitoring, where JSON has become the de facto interchange format for machine-parseable event streams [9, 12, 20, 30]. Tracers populate a fixed set of fields in this payload (e.g., file handles and byte counts in I/O events), while users attach annotations through instrumentation APIs (e.g., epoch and step in DL training loops). Despite applying compression, workloads typically scale to thousands of nodes over days of training. Traces fragmented this way

[1]Named after Horatius Cocles, who held the Tiber bridge to defend Rome, echoing Horatio's role bridging trace management and analysis while keeping raw traces intact.

may reach terabytes of uncompressed data per campaign, forcing practitioners to store, query, and correlate them across tools and layers without a schema known at write time.

2.2 Trace Analysis Stack

A typical trace analysis pipeline decompresses input files, parses each event, filters by predicate, aggregates over groups, and either reports summary metrics or hands the result to a visualization tool. Each stage contributes to end-to-end cost: decompression and JSON parsing dominate when events are scanned in bulk, while filter and aggregate dominate once predicates become highly selective. To shift this cost profile, practitioners often convert raw traces into columnar formats such as Apache Parquet [18], which organize data into row groups with per-column statistics, letting query engines prune row groups whose statistics fall outside the predicate range and read only the columns the query touches. This makes columnar layouts well-suited for queries that touch a small fraction of events or columns, but the conversion imposes a one-time storage and compute cost and requires committing to a fixed schema, which is awkward for traces whose payload is unknown at write time (§2.1). As a result, practitioners face a trade-off between full scans on raw traces and the schema engineering plus storage cost of conversion, neither satisfactory once trace volumes reach the terabyte regime.

2.3 Indexing and Execution Primitives

Querying raw traces efficiently requires bridging three gaps that their layout does not address natively: random access into compressed data, selectivity without full scans, and concurrent execution across thousands of fragmented files. For random access into gzip-compressed streams, sync-point indexing (e.g., zran [31]) records decoder state at periodic offsets so subsequent reads can resume from any checkpoint without scanning from the start. For selective filtering, Bloom filters [55] provide probabilistic membership testing with no false negatives, allowing chunks lacking a queried value to be skipped, while per-chunk min/max statistics enable analogous range pruning. For persistent storage of these structures, embedded databases (e.g., SQLite [27], RocksDB [41]) offer compact on-disk layouts and bulk-load paths suited to large index footprints. Scaling these stages across the many fragmented files of a typical trace dataset relies on thread- or process-level parallelism, since gzip decompression and JSON parsing dominate the cost of processing each event. Zero-copy buffer protocols (e.g., Apache Arrow [2] for columnar batches, POSIX shared memory [26] and Python's `memoryview` [3] for raw buffers) then let those parallel workers exchange results across language and process boundaries without serialization overhead.

3 Motivation

Horatio addresses the three limitations from the introduction: queryable raw traces without format conversion, high analysis throughput beyond general-purpose frameworks, and data-skipping on targeted queries over raw traces. **Figure 1** illustrates the core tension by plotting query throughput against storage footprint (normalized to raw size) for six trace-management approaches on a synthetic trace of ~1.2 billion events (17 GB compressed, 243 GB uncompressed). As shown in Figure 1, raw-compatible tools such as DFTracer-Utils (DFTUtils) and Perfetto stay at 1.0× storage but deliver fewer than 20 M events/s, since their parsers traverse compressed text sequentially. Conversely, converting to a binary format alongside the raw files pushes storage toward 2×: the flattened Parquet variant (ParquetFlat) reaches ~94 M events/s and a persistent DuckDB database reaches ~194 M events/s, both at ~1.9× storage, while the nested Parquet layout remains below 5 M events/s due to per-record deserialization cost. This reveals a fundamental gap that Horatio targets, occupying the high-throughput, low-storage region of Figure 1.

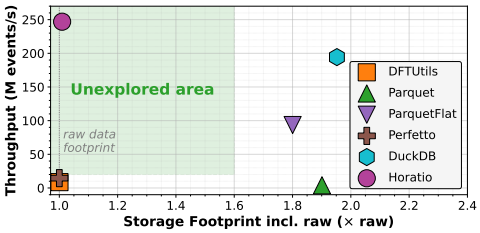


Figure 1: Storage-throughput trade-off in trace management. *Raw-compatible tools deliver <20 M events/s at* 1.0× *storage, while binary formats (Parquet variants, DuckDB) reach up to 194 M events/s only at ~1.9× storage. Horatio is the only point in the high-throughput, low-storage region. Throughput measured on a selective range query; details in §5.2.2.*

Moreover, ParquetFlat is not drop-in. Tracing libraries emit arbitrary annotations, so the schema is unknown at write time and varies across event types within a single file (e.g., I/O events and user annotations carry different metadata). Users must therefore either explode column counts when unioning incompatible schemas, or rebuild Parquet files whenever the schema shifts.

Conversion to a queryable format does not eliminate analysis bottlenecks. Pandas performs in-memory analytics, so workloads exceeding RAM must be chunked or fail outright [25]. Dask extends Pandas across nodes through task-parallel execution, but its scheduler pays roughly 200 μs of overhead per task and serializes the task graph for distribution [4, 34], with degradation reported around 100K tasks and 6 GB scheduler memory at 1.5M tasks [6, 21]. Trace workloads, which fragment into millions of small chunks at terabyte scale, hit both ceilings.

Selective queries on raw traces also pay an I/O fan-out cost. Because events are laid out by emission time, a predicate-bounded query touches hundreds of per-rank files, each contributing only a small fraction. On parallel file systems such as Lustre [13, 35] or GPFS [59], where metadata operations route through a shared metadata service, this fan-out turns selective queries into metadata-bound workloads with most decoded data discarded by the filter.

Together, these observations motivate a framework that indexes raw traces in place, sustains high-throughput analysis through a native execution path, and reorganizes events into predicate-defined groups so targeted queries skip irrelevant files.

4 Horatio

This section outlines the design of Horatio. Horatio is a raw trace management framework that addresses the three limitations mentioned in Section 3 through five components: *Reader*, *Indexer*,

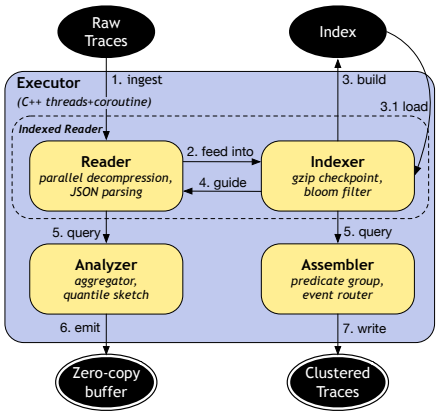


Figure 2: Design of Horatio. *Horatio's design enables selective querying, high-throughput analysis, and physical clustering of events directly on raw traces without format conversion.*

Analyzer, *Assembler*, and *Executor*, as illustrated in **Figure 2**. Raw traces are ① ingested by *Reader*, which performs parallel decompression and JSON parsing, and ② streamed into *Indexer*. *Indexer* ③ builds index structures (gzip checkpoints, bloom filters, chunk statistics) on first encounter, or loads them from a prior index on subsequent runs, then ④ guides *Reader* to skip irrelevant data, together forming the Indexed Reader. Downstream, *Analyzer* and *Assembler* ⑤ query the Indexed Reader: *Analyzer* ⑥ emits aggregated results into a zero-copy buffer for downstream Python consumption, while *Assembler* ⑦ writes physically reorganized traces clustered by predicate. *Executor*, a coroutine-backed C++ thread pool, coordinates concurrent execution across all four data-plane components.

4.1 Reader

Reader addresses the foundational challenge of accessing raw gzip-compressed traces at the throughput required by downstream components. Without parallelism, a single gzip-compressed NDJSON file decompresses sequentially, throttling every consumer that follows. *Reader* resolves this by combining two parallelism axes that compose independently: inter-file and intra-file.

Inter-File Parallelism. Files within a workload run independently across worker threads, yielding the cheapest parallelism axis since no synchronization is needed between files. Profilers like DFTracer typically emit one gzip-compressed NDJSON file per rank, so a workload with hundreds of ranks produces hundreds of independent decode units. *Reader* dispatches these to *Executor*'s thread pool and consumes them as they complete, decoupling per-file decode latency from end-to-end progress.

Intra-File Parallelism. *Reader* parallelizes within a file through one of two paths depending on whether an index is available. Without an index, multi-member files split along member boundaries, since each member is an independent zlib stream that decodes without crossing state, allowing any subset of members in any order. Runtime formats (e.g., DFTracer) adopt this deliberately, appending a new member at each buffer flush so post-hoc readers parallelize without first building an index. Once an index exists, *Reader* switches to zran-style [31] checkpoints recorded by *Indexer* at user-configurable intervals (32 MB uncompressed by default), capturing the bit offset and zlib state needed to resume decompression regardless of member layout. *Reader* divides each file into checkpoint-bounded chunks decompressed in parallel by seeding zlib with the captured state, also enabling random-offset reads when *Indexer* selects chunks against a query predicate (§4.2).

JSON Parsing. Decoded chunks contain NDJSON lines. Then, *Reader* parses them incrementally with simdjson [50], a SIMD-accelerated parser, advancing record-by-record without buffering full files. Each event is exposed as a non-owning view over the decoded buffer, avoiding per-event allocation. The parser accepts arbitrary keys without a fixed schema, matching the dynamic structure that profiling tools typically emit (§3).

4.2 Indexer

Indexer addresses the lack of selective querying by building a sidecar index that lets queries skip irrelevant data before any decompression takes place. Conventional approaches address selectivity through format conversion, which carries two costs: a storage tax from dual retention and a fixed schema that breaks under evolving instrumentation (§3). *Indexer* avoids this through three design choices: a single-pass build sharing the read path with *Reader*, all artifacts sharing decompression's chunk granularity, and user-configurable dimensions tracked at build time rather than baked into the trace format.

Single-Pass Build. *Indexer* runs as a streaming consumer of *Reader*, accumulating its three artifact classes (gzip checkpoints, predicate filters, range statistics) as decoded events flow through. At every checkpoint boundary it commits the current chunk's filters and statistics to the backing store and resets the per-chunk state for the next interval. *Indexer* also supports incremental updates, extending the existing index with only the newly added files without re-processing previously indexed data.

Bloom Filters. *Indexer* maintains per-dimension bloom filters for each chunk, populated automatically during the build from all event fields encountered, including those nested inside event payloads, with no upfront schema declaration required. Each filter uses the split-block Bloom design [32] (the same construction adopted by Apache Parquet [19]), where every insert and query touches only a single 256-bit cache-line-aligned block, keeping membership tests fast even at chunk-level granularity. At query time, predicates test against the per-chunk filter, and chunks ruled out skip decompression entirely. The dimension set is configurable, and the default already accommodates the dynamic structure of profiling traces without schema engineering.

Chunk Statistics. For predicates that require range information rather than membership, *Indexer* records per-chunk min/max values for ordered dimensions and approximate distributions for numeric ones. Range pruning eliminates entire chunks for time- or value-bounded queries, while distribution sketches answer aggregate queries such as percentiles or histograms directly from the index without touching the trace.

RocksDB Backend. *Indexer* persists all index artifacts in a per-trace RocksDB [41] key-value store. We select an LSM store over

a single-writer B-tree engine such as SQLite [27] (used by dftracer-utils and zindex), whose serialized writes bottleneck parallel index construction (§5.2.2): RocksDB instead admits lock-free concurrent writes, per-chunk keying, an offline SST bulk-ingest path, and a merge operator for aggregation, matching our write-once, read-many index workload. Each artifact class lives in its own logical keyspace, with keys composed from the chunk identifier (and dimension where applicable), so a query touches only the per-chunk artifacts it needs. For distributed builds, the coordinator partitions input files across workers using greedy Longest-Processing-Time-first (LPT) bin-packing [44] to balance per-worker load by file size. Each worker then spills its artifacts as RocksDB shards that the coordinator bulk-ingests, skipping the memtable so ingestion completes as a metadata operation rather than a series of synchronized writes.

4.3 Analyzer

Analyzer addresses analysis throughput bottlenecks by moving event-level computation out of Python and into a native C++ pipeline. Conventional approaches built on Pandas or Dask are bottlenecked at terabyte scale by in-memory limits and scheduler overhead (§3). *Analyzer* resolves this through three design properties: end-to-end native C++ event processing, streaming aggregation that emits results incrementally rather than materializing full datasets, and zero-copy C++/Python data exchange.

Streaming Aggregation. *Analyzer* groups events by a composite key (e.g., process id, category) and slices them into time windows, mirroring the main-view abstraction of WisIO [69]. Per-window hash maps commit to a shared RocksDB store via its merge operator [41] at window close, avoiding the all-at-once dataframe materialization that Pandas-based pipelines require.

Mergeable Sketches. For numeric quantities such as event durations, *Analyzer* maintains DDSketch [51] digests per group with bounded relative error (1% by default) that merge associatively across workers without a coordination barrier. Memory stays bounded through a collapsing dense store that absorbs the lowest bins once the active key range exceeds the bin limit. The same mergeability extends to count and histogram aggregates, combining per-window results in any order at window close.

Zero-Copy Python Exchange. *Analyzer* produces Apache Arrow [2] record batches in C++ that Python consumers access directly through Arrow IPC, without buffer copies or cross-language serialization. Downstream tools such as Pandas, Polars, or DuckDB consume the tables natively without further conversion.

4.4 Assembler

Assembler addresses the lack of physical clustering, where write-order layouts force targeted queries to fan out across scattered reads. Repartitioning via dataframe shuffles or Parquet partitioning addresses this but loses raw-format compatibility. *Assembler* reorganizes the on-disk layout once and preserves the raw-format contract through three design properties: predicate-defined groups specified at extraction time, format-preserving NDJSON output with tunable chunk sizes, and a lossless transformation reversible to the original, so the reorganized files can act as the canonical copy without retaining both layouts.

Predicate-Based Clustering. Users define groups by query predicates, with each group becoming a target output partition (e.g., one group filtering on event category, another on rank range). *Assembler* consults *Indexer* (§4.2) to identify which chunks contain events matching each predicate, skipping chunks that no group requires. Within qualifying chunks, an *event router* evaluates the predicate per event and dispatches it to the matching target group, with each group's events ending up in contiguous storage for sequential reads with strong cache locality.

Format-Preserving Output. Output remains in the input's gzip-compressed NDJSON format, so downstream tooling consumes it unchanged. By default, *Assembler* runs in lossless mode, preserving every event line verbatim and recording enough provenance per group to rebuild any original file on demand, so users do not need to retain the originals alongside the reorganized output. Because clustering only regroups whole event lines and never merges, dedups, or delta-encodes events, every field including the exact timestamp is byte-preserved, avoiding any precision-versus-compression trade-off. The reorganized layout is not smaller than raw (1.23–1.48×, §5.4) but is physically ordered for skip-friendly access. A lossy mode is also supported for users who want to trim trace size from excessive profiling.

Tunable Output Chunking. *Assembler* writes each group's events into output files of configurable size (default 256 MB of uncompressed data per chunk). On Lustre, *Assembler* automatically queries the stripe size and aligns flush boundaries with it, so writes hit one stripe at a time without users tuning the chunk parameter manually, avoiding the small-file metadata costs discussed in §3.

4.5 Executor

Executor provides the concurrency and execution backbone shared by all four data-plane components, where a naive thread-per-file or task-per-chunk approach quickly hits OS-level limits at the parallelism trace workloads demand. *Executor* resolves this through three design properties: C++20 coroutines for lightweight cooperative tasks, async channels for pipeline-style composition between components, and MPI integration for cross-node execution.

Lazy Coroutines. *Executor* uses C++20 stackless coroutines configured to suspend immediately on creation, so each coroutine becomes a unit of scheduled work rather than starting eagerly when called. This lets the scheduler decide when each task begins, prioritizing tasks whose dependencies are ready and stalling tasks waiting on I/O without consuming a thread, keeping the active task set bounded even at trace-workload scale.

Channel-based Compositions. Components communicate through bounded async channels, with each channel acting as a lock-free buffered queue between producer and consumer coroutines. *Reader* feeds decoded events into a channel drained by *Indexer*, *Analyzer*, or *Assembler*, allowing the four data-plane components to be composed into different pipelines without changing their internal logic. Bounded channels also provide backpressure naturally: when a consumer falls behind, the channel fills and the producer suspends until space is available, decoupling component lifetimes without requiring explicit flow control.

MPI for Distributed Execution. For workloads that exceed a single node's capacity, *Executor* exposes an MPI-backed mode where

each rank runs an independent coroutine pool over its partition of the input files. Within a rank, all state (channels, per-chunk artifacts, aggregation maps) stays local, so the data-plane phase requires no cross-rank synchronization. Cross-rank aggregation proceeds through one of two paths: small reductions (e.g., DDSketch merges) use MPI collectives, while large index outputs spill to local RocksDB shards that the coordinator bulk-ingests, avoiding the cost of shipping large payloads over MPI (§4.2).

4.6 Implementation Details

Horatio is implemented in approximately 175,000 lines of C/C++ and Python across the data-plane components, runtime, and Python bindings. We describe the key implementation aspects below.

Build and Dependencies. The data plane is written in C++20 (requiring coroutines and concepts) and exposed to Python through a C extension module packaged for cross-platform wheels. Horatio relies on RocksDB [41] for the index store, zlib-ng [70] for vectorized gzip decompression, simdjson [50] for JSON parsing, nanoarrow [15] for Arrow record-batch construction, moodycamel's ConcurrentQueue [39] for the channel substrate, and DataDog's DDSketch reference [51] for quantile sketches.

Runtime Internals. I/O routes through a runtime-selected backend: on Linux 5.1+, a raw-syscall `io_uring` [7] ring (256 entries, batched submission) drained by a dedicated completion thread with no kernel-side polling (neither `SQPOLL` nor `IOPOLL` and no `O_DIRECT`), falling back to an `epoll` (Linux) or `kqueue` (macOS/BSD) thread pool, and a portable thread pool elsewhere. Because C++20 coroutine frames are heap-allocated by default and short-lived task churn would contend on the allocator at scale, Horatio uses a process-global object pool that recycles frame slots across tasks.

MPI and Dask Integration. For Python users who prefer Dask-based orchestration, a thin shim presents Horatio's per-rank kernels as Dask tasks while keeping all event-level work in C++ via the zero-copy buffer (MPI is optional and gated by a CMake flag).

5 Evaluation

This section evaluates Horatio's effectiveness in managing raw traces for scientific and AI workloads. We address three questions aligned with our contributions:

(1) **Storage-efficient querying.** Can lightweight indexing over raw traces deliver fast queries at minimal storage overhead?
(2) **End-to-end analysis performance.** Can a native C++ analysis pipeline with zero-copy Python exchange yield higher throughput than a trace analysis pipeline at the same parallelism?
(3) **Physical data clustering.** Can physical trace clustering further reduce end-to-end analysis time on top of an already optimized analyzer?

The evaluation proceeds in five parts: experimental setup; indexing build cost, storage overhead, and selective query performance against Parquet-based baselines; analysis performance against DFAnalyzer on real and synthetic traces; the benefit of physical clustering; and scaling across cores and nodes.

5.1 Experimental Setup

The full source, experiment scripts, and workload configurations are publicly available for reproduction.[2]

5.1.1 Cluster Configuration. All experiments were conducted on the Corona cluster at Lawrence Livermore National Laboratory. Each node has dual AMD Rome CPUs with 48 cores and 96 hyperthreads, 256 GB of memory, a 1.5 TB local SSD, and a peak single-CPU memory bandwidth of 153 GiB/s. Nodes are interconnected via HPE Slingshot 11 and a shared Lustre parallel file system with a peak aggregate bandwidth of 240 GB/s. Lustre experiments use the maximum stripe count (all OSTs) with a 32 MB stripe size.

5.1.2 System Configuration. Horatio and all baselines run with every hyperthread available and no explicit core pinning; Horatio sizes its C++ pool to the hardware concurrency plus one I/O completion thread. Since Corona's kernel predates `io_uring`, Horatio uses its `epoll` thread-pool backend, which dispatches each read to a dedicated I/O thread pool issuing blocking buffered `pread` calls (`epoll` cannot poll regular files). The `io_uring` path of §4.6 applies on Linux 5.1+ and `kqueue` on macOS/BSD. Its RocksDB index store uses ZSTD-9 SST compression, a 256 MiB memtable, disabled WAL, and a bulk-ingest path via sorted SST files ingested without the memtable (block cache at RocksDB's default).

5.1.3 Trace Datasets. We evaluate Horatio's performance on one synthetic trace and four real workload traces spanning AI and scientific workloads profiled by DFTracer. **Table 1** summarizes the number of generated traces, compressed and uncompressed sizes.

Synthetic (SYN) is generated using our trace generator modeled after UNet3D distributed training. The configuration uses 256 ranks across 500 epochs of 500 steps each, producing one gzip-compressed NDJSON file per rank with a known event mix covering training steps, periodic checkpoints, and validation phases.

Montage [58] is collected from the Montage astronomical mosaicking workflow processing FITS images from the Two Micron All-Sky Survey (2MASS). The workflow chains 10 applications that reproject and rectify input images onto a uniform pixel grid. The trace captures per-task I/O across all stages over 5 iterations of a 2.5 GB 2MASS dataset on 32 nodes with 32 processes per node.

UNet3D [37] is collected from a PyTorch UNet3D training run following the MLPerf Training [52] configuration, which uses the No New-Net [46] architecture for kidney tumor segmentation on the KiTS19 [8, 45] dataset. The trace spans 20 epochs across 128 AMD MI300A APUs, where each GPU process and its data loader worker each emit a single gzip-compressed NDJSON file.

Llama [63] is collected from a PyTorch + DeepSpeed [57] pretraining run of a Llama-1B model on the pretokenized OpenHermes-2.5 [1] conversational dataset. The trace spans 20 epochs across 512 AMD MI300A APUs under DeepSpeed ZeRO-0 with one data loader worker per rank.

h5bench [36] is collected from an asynchronous write workload using the HDF5 Async VOL [61] connector on 8 nodes with 64 processes per node (512 processes total). We use a contiguous memory and file pattern with dimensions (256, 128, 128) in file-per-process mode, generating 1.3 TB of output. The trace captures

[2] https://github.com/rayandrew/horatio-artifact

Table 1: Trace datasets used in the evaluation.

Trace	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

POSIX, MPI-IO, and internal HDF5/MPI/h5bench calls in full, producing a 2.2 TB uncompressed footprint. This footprint serves as the scaling stress-test in §5.5.

5.1.4 Baselines. We compare Horatio against two categories of baselines, one for trace format and one for analysis backend.

Trace format baselines cover the main alternatives for storing and querying raw traces. All Parquet variants are built using PyArrow [23] with `zstd` compression and 64 MB row-group batches, and retain the original raw files alongside the converted output. **Parquet** converts each raw NDJSON trace into a Parquet file where any nested JSON object in the traces is serialized as a JSON column string. **ParquetFlat** instead flattens every nested JSON field into its own top-level column, allowing Parquet's column projection and predicate pushdown to apply to every field and representing the most favorable Parquet configuration. **Perfetto** [20] represents the full-ingest analysis path used by Perfetto's visualization tooling, which loads the entire trace into an in-memory database before querying. **DuckDB** [56] is an embedded analytical SQL database that we evaluate in two modes. **DuckDB (persistent)** ingests every NDJSON trace once via `CREATE TABLE ... AS SELECT FROM read_json(...)` into an on-disk database that subsequent queries open read-only, producing a persistent columnar representation comparable to Parquet's. **DuckDB (in-mem)** skips the build step and reads the raw NDJSON traces directly at query time, materializing them into an in-memory database before evaluating the predicate. The two modes bound DuckDB's trade-off: persistent pays a one-time build and storage cost for parse-free scans, while in-mem produces no on-disk artifact but re-ingests per query. Both run with default configuration, passing only the format hints required for gzip-compressed NDJSON input (`format='newline_delimited'`, `compression='gzip'`, `ignore_errors=true`). DFTracer also comes with dftracer-utils [11] (**DFTUtils**) that provides lightweight indexing on top of SQLite [27] over gzip-compressed traces to enable parallel decompression over gzip chunks.

Analysis backend baseline. DFAnalyzer [10], an implementation of WisIO [69], an enhanced version of the original DFAnalyzer shipped with DFTracer [40], built on Dask [5], is the analysis pipeline where both orchestration and event-level computation run in Python under Dask. DFAnalyzer uses dftracer-utils [11] as its gzip indexer to expose gzip chunks as parallel units of work across Dask tasks. We also integrate Horatio into DFAnalyzer, keeping Dask for orchestration but offloading C++-friendly kernels to its C++ backend via zero-copy exchange, isolating the analyzer's speedup while showcasing cross-language interoperability.

5.1.5 Metrics. We report three performance metrics. **Throughput** is million events processed per second, counting events after format decoding (post-decompression for compressed formats). **Bandwidth** is GB/s over the *uncompressed* data volume, directly comparable across formats with different compression (gzip for raw NDJSON, `zstd` for Parquet). **Wall-clock time** is the end-to-end pipeline duration, used when reporting end-to-end results. Every build and query measurement is repeated 5 times. Bars report the mean across the 5 runs, and error bars are derived from the population standard deviation (σ) of wall-clock time.

5.2 Storage-Efficient Querying

To evaluate the storage-throughput performance of Horatio compared to format conversion, we run a single-node experiment on local SSD using all 96 hyperthreads, comparing Horatio, dftracer-utils, Parquet, ParquetFlat, Perfetto, and both DuckDB modes (persistent and in-mem) on Synthetic traces. Each Parquet variant is built per trace file, producing multiple Parquet files that mirror the per-rank layout of the raw traces.

5.2.1 Schema and Storage Overhead. **Figure 1** (§3) summarizes the storage-throughput trade-off across all formats. Horatio reaches ~167 M events/s at 1.01× raw storage (a 152 MB index alongside the 17.8 GB compressed trace), while Parquet variants and persistent DuckDB sit at ~1.8–1.95× storage and dftracer-utils and Perfetto remain near 1.0× but below 20 M events/s. The two Parquet variants also differ in schema. Naive Parquet keeps a compact seven-column layout (`timestamp`, `ph`, `name`, `pid`, `ts`, `dur`, and a single `args` JSON-string column). The `args` object is heterogeneous: it carries built-in fields from interceptors (e.g., `fhash`, `ret`, `count`, `size`) and user-annotated metadata (e.g., `epoch`, `step`) registered through DFTracer's `update_metadata` API. ParquetFlat lifts every observed `args` key into its own top-level column, expanding to 30+ columns and requiring an upfront pass to discover the field set. This is impractical in practice since trace schemas evolve as users add annotations or instrument new code paths; this is why, for instance, DFTracer adopts an extensible JSON line format. Persistent DuckDB inherits the same schema rigidity through `read_json` [65]'s 20k-row sample, silently dropping late `args` fields. `sample_size=-1` recovers them but sacrifices ingestion throughput and still freezes the schema at build time. The result lands at ~1.95× raw storage, on par with ParquetFlat, and shares ParquetFlat's brittleness to schema drift since any new annotation requires re-ingesting the trace.

5.2.2 Selective Query Performance. **Figure 3** reports query-and-materialize throughput across three representative access patterns that each skip more than 99% of events, with per-format index build time overlaid on the right y-axis: range pruning on a temporally ordered field (`epoch_filter`, panel (a)), a partition-aligned single-file scan (`rank_filter`, panel (b)), and a compound high-cardinality predicate (`node_file_io`, panel (c)). Panel (d) summarizes each format's throughput spread (min-to-max range, arithmetic mean as center dot). Each query applies its predicate and materializes matching events, so column count contributes to decode cost even when the predicate touches one field. For fair comparison, all formats are measured end-to-end in Python and materialize matched

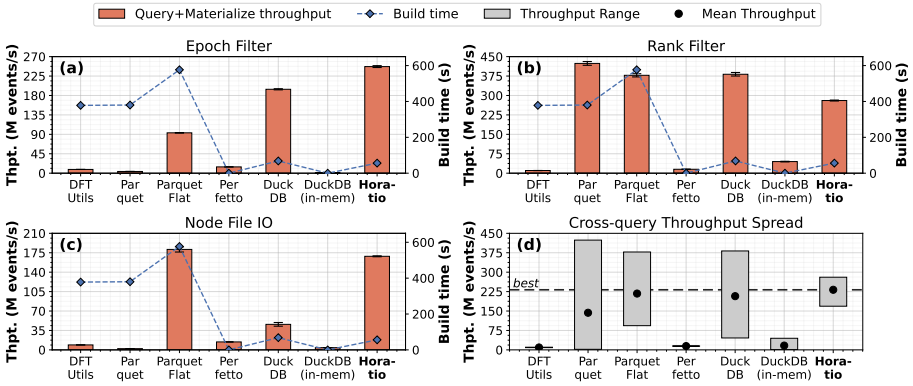


Figure 3: (a)–(c) Query+materialize throughput with build time and (d) cross-query throughput spread on Synthetic traces. *Horatio builds the fastest at 57 s (up to 10.1× faster than other formats) and sustains 169–281 M events/s across all three queries, with the highest cross-query mean (232 M events/s) and tightest spread among high-throughput formats.*

rows into Arrow Tables via zero-copy (Parquet via PyArrow [23], DuckDB via native Arrow, Horatio via its Python binding).

Build cost. The right-axis line in panels (a)–(c) reports per-format build time, with all builds running at 96 hyperthreads (DuckDB internal threads, Horatio via its C++ pool, the Parquet variants and DFTUtils via a 96-worker process pool over 256 input files). Horatio writes its index in 56 s, faster than any format, with persistent DuckDB next at 68 s and DFTUtils (378 s), naive Parquet (380 s), and ParquetFlat (577 s), using vectorized gzip decompression and JSON parsing while building a small index over the raw traces. DuckDB stays close because it ingests at high throughput by inferring its schema from a 20k-row sample (§5.2.1), at the cost of silently dropping any `args` field that appears only after the sample window. The Parquet variants have no equivalent mechanism since PyArrow's `pyarrow.json.read_json` rejects mixed-type columns even with an explicit schema [24], forcing per-file Python ingestion with an extra key-discovery pass for ParquetFlat. DFTUtils lands in the same band despite producing a far smaller artifact, bound by Python text parsing over single-threaded SQLite writes. DuckDB (in-mem) and Perfetto have no build phase and sit at zero on the right axis, paying their ingest cost at query time instead.

`epoch_filter`: range pruning. This query selects all events with `epoch == 0` to isolate a single training epoch, a common DL analysis pattern. It isolates skip granularity since both Horatio and ParquetFlat expose `epoch` as a first-class field (a chunk-level statistic in Horatio, a column in ParquetFlat). Horatio leads at 247 M events/s (53 GB/s) by pruning at gzip-chunk granularity and decompressing only chunks that survive the predicate, ahead of persistent DuckDB at 194 M (41 GB/s) and ParquetFlat at 94 M (20 GB/s) whose block-level pruning is coarser. Naive Parquet collapses to 4 M events/s (0.85 GB/s) since `epoch` is buried in its `args` JSON column without pushdown, and DuckDB (in-mem, 0.96 M), Perfetto (15 M), and DFTUtils (8.7 M) trail since they re-ingest or scan the full trace without chunk-level skipping.

`rank_filter`: partition-aligned scan. This query selects all events with `pid == 1000` to isolate a single rank (e.g., straggler inspection). Since each input file is per-rank, every format reduces to a single-file scan that isolates raw decode throughput rather than skip granularity. Naive Parquet leads at 424 M events/s (90 GB/s), with persistent DuckDB and ParquetFlat tied just behind at 382 M (82 GB/s) and 378 M (81 GB/s), all three exploiting pre-decoded columnar layouts where `pid` is read directly without touching nested JSON. Horatio reaches 281 M events/s (60 GB/s), 1.51× behind the leaders, since it must still decompress and JSON-parse each surviving gzip chunk rather than read a prebuilt column. DuckDB (in-mem, 45 M), Perfetto (15 M), and DFTUtils (10 M) trail because they neither prune nor benefit from a pre-decoded columnar layout.

`node_file_io`: compound bloom predicate. This query filters on both `hostname` and `file` (e.g., `host == 'node-0'` AND `file == '/ckpt/...'`), a highly selective compound predicate over `args`-nested fields that mirrors per-file I/O diagnosis on a node. ParquetFlat leads at 181 M events/s (39 GB/s) by combining row-group pushdown on its flattened `hostname` and `file` columns with light row materialization, with Horatio 1.07× behind at 169 M events/s (36 GB/s) where per-chunk bloom filters skip chunks that cannot match the predicate, but surviving chunks still incur decompression and JSON parsing. Persistent DuckDB drops to 46 M events/s (10 GB/s), an 8.3× regression from `rank_filter` since its row-group zonemaps [64], whose effectiveness depends on column ordering, likely prune less when `hostname` and `file` interleave across row groups. Naive Parquet drops to 2.2 M events/s (0.47 GB/s) because both fields live inside its serialized `args` JSON column, and DuckDB (in-mem, 4.0 M), Perfetto (14 M), and DFTUtils (8.9 M) trail since they re-ingest or scan the full trace without selectivity on JSON-buried fields.

Crossover: when Horatio wins and when it loses. The three queries expose a simple trade-off. Horatio decodes each surviving byte about 1.51× slower than a columnar format, since it must decompress and JSON-parse each gzip chunk while Parquet or DuckDB reads a pre-decoded column. This gap is measured on `rank_filter`, where the per-rank layout lets *no* format skip anything. Horatio offsets it by skipping more aggressively: it prunes at gzip-chunk granularity with per-chunk bloom filters, whereas Parquet prunes only at the coarser row-group level and only on

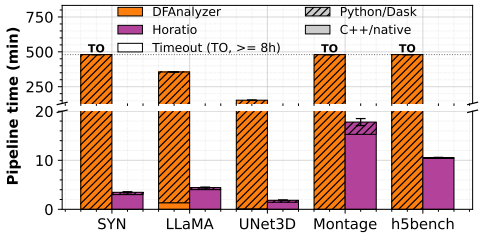
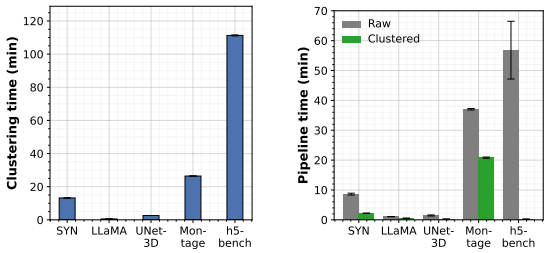


Figure 4: End-to-end pipeline time on 8 nodes. *Horatio's native-backed Python pipeline completes all workloads in 2–18 min, while DFAnalyzer's Python-compute pipeline either takes hours or hits the 8 h budget (TO).*



(a) Clustering Cost **(b) Pipeline Time**

Figure 5: Clustering cost and end-to-end speedup on a single node. *Pipeline runs 1.8–5.5× faster on the clustered layout across four workloads and 230× on h5bench, with a one-time cost of 0.5 min–1.85 h paid once per trace and reused across subsequent analyses.*

materialized columns. So Horatio wins whenever its finer pruning lets it decode less data to cover the 1.5× penalty. Assuming f is the fraction of data still decoded after pruning, Horatio overtakes the columnar baseline once $f_H/f_{col} \lesssim 1/1.51 \approx 0.66$, i.e., when it decodes under two-thirds as much surviving data. They fall at distinct points along it. On `rank_filter` nothing can be pruned (a single-file scan), so the penalty is fully exposed and columnar wins by 1.51×. On `node_file_io`, ParquetFlat's flattened columns prune almost as finely as Horatio's bloom filters, so the two stay within 1.07×. On `epoch_filter`, Horatio's chunk-level skipping is far finer than row-group pruning, so it leads the best columnar baseline by 1.27× (and ParquetFlat by 2.6×). The practical rule: partition-aligned full scans are the one pattern where format conversion buys a modest (∼1.5×) win, while every selective pattern with exploitable structure favors Horatio's single-copy index.

Implications for format conversion. Rankings reflect format-plus-runtime as a unit (PyArrow, DuckDB, Horatio's runtime), and each columnar format still pays ∼1.8–1.95× raw storage and re-conversion on schema evolution (§5.2.1). Figure 3(d) shows Horatio as the highest-mean, tightest-spread high-throughput format, on a single gzip copy at 1.01× storage.

Takeaway 1. *Horatio delivers high across-query throughput (232 M events/s mean, lowest variability) at a 56 s build and ∼1.0× storage.*

5.3 End-to-end Analysis Performance

To evaluate analysis throughput, we compare the original Dask-based DFAnalyzer against a modified DFAnalyzer that retains Dask for orchestration but offloads event-processing kernels to Horatio's C++ backend. Both run on 8 nodes with 4 Dask workers per node and 24 threads per worker (fully utilizing each node's 96 hyperthreads) under an 8-hour wall-clock budget, over four real traces (Montage, UNet3D, Llama, h5bench) and one synthetic trace. We use DFAnalyzer's POSIX preset (per-file I/O metrics) for Montage, Synthetic, and h5bench, and the DL preset (per-layer training metrics across data loader, reader, training, and checkpoint events) for UNet3D and Llama.

Figure 4 reports wall-clock pipeline time per workload (lower is better), with each bar split into Python/Dask and native phases. Across 5 runs we observe under 5% variability in wall time. Horatio completes all five pipelines within 2–18 min, routing event-level compute through the native phase while keeping dataframe-shaped stages in Python/Dask. DFAnalyzer's native phase is confined to DFTUtils-based gzip parsing, and it finishes only UNet3D (∼2.5 h) and Llama (∼6 h) before timing out on other larger workloads. On the two workloads where both complete, Horatio delivers 80–83× speedup at matched parallelism.

The original DFAnalyzer parses gzip chunks via DFTUtils but materializes every event into Pandas dataframes, hitting two compounding bottlenecks. First, it issues one Dask delayed task per 4 MB chunk per file: Llama alone produces ∼21K graph layers and a 200+ MiB serialized task graph, and the scheduler spends tens of minutes deserializing and walking it before dispatching any tasks. Second, event-level work then runs in Python with per-row dispatch and Pandas materialization, so per-worker memory grows with partition size rather than streaming through. Horatio addresses both by routing event-level compute (decompression, JSON parsing, predicate evaluation, streaming aggregation) through a native thread pool with LPT-balanced coarse-grained tasks that build a local RocksDB shard per worker, leaving Dask's scheduler far fewer tasks while keeping dataframe-shaped stages in Python/Dask where Pandas/Arrow are most efficient. These bottlenecks also show in resource use: because Horatio streams events through bounded channels, its peak resident memory stays at 0.4–3.8 GB across all workloads, against DFAnalyzer's 9.5 GB at pipeline scale (rising to the ∼110 GB that drives its h5bench OOM, §5.5.2), and DFAnalyzer saturates every core yet stays slower because per-row Python dispatch trades CPU for little throughput.

Takeaway 2. *Horatio's native-backed pipeline finishes all five workloads in 2–18 min and runs 80–83× faster than DFAnalyzer, which times out on three of them.*

5.4 Trace Clustering

To evaluate how trace clustering affects end-to-end analysis time, we run the DFAnalyzer pipeline backed by Horatio on a single node with 96 threads over local SSD, comparing the original traces against a clustered layout produced by *Assembler* that groups events into `I/O` (POSIX/STDIO) and `others` categories. Using the same analyzer on both layouts isolates the speedup as a clustering effect rather than an analyzer change. Clustering runs once in lossless mode (§4.4) and lands at 1.23–1.48× raw size while replacing the originals on disk. The reorganized layout remains reconstructible

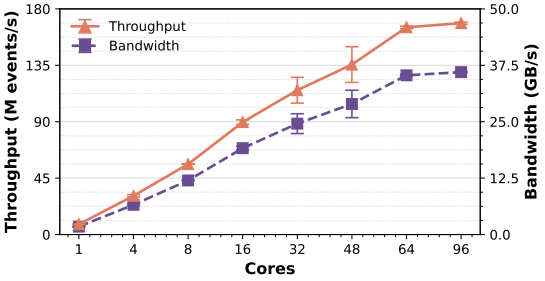
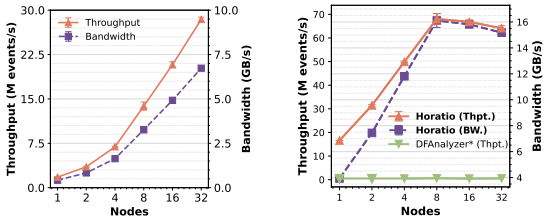


Figure 6: Core scaling on Synthetic traces. *Horatio scales near-linearly to 32 threads (14×) and reaches 21× at 96 threads.*



(a) MPI Pipeline (b) Dask Pipeline

Figure 7: Node scaling on h5bench. *Both Horatio backends complete the 2.2 TB workload at every node count, with MPI/C++ scaling from 1.8 to 28 M events/s (16×) and Dask peaking at 68 M events/s by 8 nodes on the preset-filtered pipeline. *Original DFAnalyzer hits OOM at every scale.*

on demand from the recorded provenance, so users do not need to retain a separate raw copy.

We pick a DFAnalyzer preset per workload that matches its typical analysis use case: POSIX for the I/O benchmarks (Montage, h5bench) and DL for the deep learning workloads (Synthetic, UNet3D, Llama). The POSIX preset reads only `I/O` and the DL preset reads only `others`, so clustering lets each preset skip events outside its target. **Figure 5** reports (a) one-time clustering cost and (b) end-to-end pipeline elapsed time on raw vs clustered layouts. Across 5 runs per workload, clustering and clustered-pipeline runs show under 8% variability, while raw-pipeline runs reach up to 17%, mostly on h5bench where the 2.2 TB scan exposes system-level noise (e.g., Lustre contention).

Clustering cost varies across workloads: Llama finishes in 0.5 min, UNet3D in 2.6 min, Synthetic in 13 min, Montage in ~26 min, and h5bench in ~1.85 h on a single node. h5bench's trace is dominated by MPI-IO and internal HDF5/MPI/h5bench calls (§5.1.3), so to separate POSIX/STDIO events from this much larger bulk, *Assembler* must scan every one of the 10.6 B events since the I/O operations are interleaved inside those calls. This one-time cost is reused across subsequent analyses.

Once clustered (Figure 5b), the pipeline runs 1.8× faster on Montage, 2.0× on Llama, 3.9× on Synthetic, 5.5× on UNet3D, and 230× on h5bench, as reading contiguous per-category regions removes seek overhead and skips irrelevant chunks. For the POSIX preset, h5bench's massive MPI-IO/HDF5 events live entirely in `others` and are skipped (230×), while Montage's events are mostly POSIX so `others` is small and the gain is modest (1.8×). For the DL preset, Synthetic and UNet3D produce sizable `I/O` groups via file reads from dataloaders, so skipping them yields 3.9× and 5.5×, while Llama's `mmap`-based dataset access bypasses the user-space POSIX hooks DFTracer instruments, leaving its `I/O` group too sparse to skip (2.0×).

Takeaway 3. *Clustering raw events by category yields 1.8–230× speedup across the five workloads.*

5.5 Scaling

We run two scalability studies. Core scaling sweeps thread counts from 1 to 96 on Synthetic at a single node over local SSD on the highly selective `node_file_io` query, isolating intra-node parallelism. Node scaling exercises h5bench's 2.2 TB footprint across 1 to 32 nodes over Lustre at one process per node with 96 C++ threads. It measures two Horatio configurations independently: the MPI/C++ aggregator (full aggregation, no preset) and DFAnalyzer backed by Horatio's Dask + C++ backend (§5.3, `I/O` preset), with the original Dask-only DFAnalyzer as a baseline for the Dask path.

5.5.1 Core Scaling. **Figure 6** reports throughput and bandwidth on the highly selective `node_file_io` query as the C++ thread count varies on a single node. X-axis is core count, the left y-axis is throughput in million events per second, and the right y-axis is bandwidth in GB/s. Throughput climbs from 8.2 M events/s at one thread to 115 M events/s at 32 threads (~14×) and continues to 165 M events/s at 64 threads as hyperthreads cover memory-access latency. All 96 hyperthreads reach 169 M events/s (21× overall), with the final step from 64 to 96 threads adding only 2% as sustained bandwidth approaches ~36 GB/s.

The efficiency profile localizes the bottleneck: parallel efficiency falls from 100% at one thread to 44% at 32 (14×), 31% at 64 (20×), and 22% at 96 (21×). Through the first socket the pipeline is compute-bound on per-chunk decompression and JSON parsing, which scales with physical cores, so throughput climbs steeply (8.2 to 115 M events/s). Past the 48 physical cores, added hyperthreads only hide memory-access latency on the index-probe path, explaining the 115 → 165 M gain from 32 to 64 threads but the near-flat 64 → 96 step once both sockets saturate. The ceiling is memory-system latency, not bandwidth or I/O: sustained bandwidth plateaus at ~36 GB/s, far below the 153 GiB/s peak, because a query this selective mostly probes per-chunk metadata and decodes only the few surviving chunks, plus a serial tail (chunk enumeration and Arrow assembly) that Amdahl-caps the last threads. Consistent with this, Horatio keeps only ~34 of 96 cores busy on this selective query, as chunk skipping leaves limited surviving-chunk work rather than saturating all cores; less selective workloads, where decode dominates, scale closer to linear.

5.5.2 Node Scaling. **Figure 7** reports throughput and sustained bandwidth on h5bench for (a) Horatio's MPI/C++ aggregator and (b) DFAnalyzer backed by Horatio's Dask + C++ backend, sweeping 1 to 32 nodes. The green line in (b) is the original Python-only DFAnalyzer (no bandwidth, as every run OOMs before completion). The MPI/C++ pipeline scales from 1.8 to 28 M events/s (16× at 32 nodes), with bandwidth climbing from 0.4 to 6.7 GB/s. Scaling is near-linear through 8 nodes (7.7×, 96% efficient) and then decays to 50% efficiency at 32 nodes (16×) as cross-node aggregation overhead

grows with rank count. This overhead is communication, not compute: after each rank scans its shard locally, the aggregator runs a global `MPI_Allgatherv` of the cross-rank gzip-member map and an `MPI_Gatherv` of per-rank partial aggregates, collectives whose cost grows with rank count and aggregation cardinality. Since per-node decode work is fixed, doubling nodes past 8 adds gather traffic (and Lustre contention) faster than useful decode, resulting in throughput rising just ∼14 to 28 M events/s over the 4× from 8 to 32 nodes.

The Dask + Horatio pipeline climbs from 17 to 68 M events/s by 8 nodes (4×, bandwidth 3.9 to 16 GB/s), then plateaus and slightly regresses (64 M at 32 nodes) as Dask's task-graph overhead caps further gains. Dask still achieves higher absolute throughput per node than MPI/C++ because the `I/O` preset filters non-`POSIX/STDIO` events out of the aggregation surface, while MPI/C++ aggregates the full event set and processes more events per chunk. The original DFAnalyzer (green line) hits out-of-memory on h5bench across 1–32 nodes, each run terminating after 5–7 h at ∼110 GB peak RSS per worker, well before the 12 h budget. Even with disk-spill configured, the task-graph and Python-side per-partition Pandas overhead (§5.3) compound to exhaust memory at this trace size.

Takeaway 4. *Horatio delivers up to 21× core, 16× MPI-node, and 4× Dask-node scaling on terabyte traces where DFAnalyzer OOMs.*

6 Related Work

6.1 Trace Formats and Indexing

Trace storage falls into structured columnar formats and per-profiler binary formats. Apache Parquet [18] provides a columnar format with row-group statistics and bloom filters for analytical queries on a fixed schema. I/O profilers ship their own formats: Darshan [60, 68] and Recorder [66] use custom binary records, while DFTracer [40] adopts gzip-compressed NDJSON for readability and tool compatibility. For random access into gzip streams, zran [31] introduced the checkpointed-index approach used by dftracer-utils [11], and zindex [42] offers similar functionality, both backed by SQLite [27]. A separate line of HPC trace compressors exploits event repetition: ScalaTrace [53] merges equivalent MPI events and stores delta times rather than absolute timestamps, and its elastic extension [67] approximates timing with histograms, trading precision for compression. OTF2 [49] keeps exact per-event timestamps losslessly but does not shrink below the full event set. Horatio's clustering is orthogonal: it regroups events byte-exact (timestamps included), trading a small storage increase for skip-friendly locality without approximating timing. Horatio builds on the indexing approach with a RocksDB-backed sidecar unifying checkpoints, bloom filters, and chunk-level statistics for selective queries without conversion.

6.2 Trace Analysis Frameworks

Existing trace analysis falls into per-profiler ecosystems and general-purpose data processing frameworks. Profiler-specific tools such as DFAnalyzer (now built on WisIO [69]) and PyDarshan [38] provide curated analysis views atop Python. General-purpose engines split between Python-resident (Pandas [17], Dask [5]) and native (Polars [54], DuckDB [56]), the latter increasingly built around Apache Arrow. Horatio focuses on raw-trace analysis with a native C++ pipeline and zero-copy Arrow exchange to Python. Rather than replacing them, it serves as a trace-aware backend they build on: Dask orchestrates, PyArrow/Polars/DuckDB consume the Arrow output, and Parquet writers archive results.

6.3 Parallel Gzip Decompression

A separate line of work targets parallel decompression of arbitrary gzip streams: pugz [47] pioneered this for ASCII text, while RapidGzip [48] generalized it via cache-based prefetching to multi-GB/s throughput. Vectorized paths such as libdeflate [33] and zlib-ng [70] accelerate per-thread decode within these schemes. Horatio integrates zlib-ng for per-thread decode and zran-style checkpoints for cross-thread parallelism. In future work, RapidGzip-style prefetching could complement it to accelerate index building.

7 Conclusion

Profiling and tracing are essential to HPC and AI workloads, but their traces grow to volumes where conventional management forces trade-offs between storage, queryability, and analysis throughput. We develop Horatio, which manages raw traces in place: building lightweight indexes, running native event-level analysis, and clustering events by predicate without leaving the raw format. A RocksDB sidecar of zran-style checkpoints, per-chunk bloom filters, and chunk statistics delivers selective queries up to 75× faster than naive Parquet at under 1% storage overhead. A native C++ backend behind Dask accelerates DFAnalyzer by 80–83× and completes terabyte-scale runs it cannot, and predicate-based clustering adds a further 1.8–230× while keeping the raw format reversible. On a 2.2 TB trace, Horatio scales to 16× (MPI) and 4× (Dask) where DFAnalyzer OOMs, and integrates as an Arrow-native backend for Pandas, Polars, and DuckDB. As future work, we plan NUMA-aware scheduling, additional profiler formats, and an indexed-trace backend for visualization tools such as Perfetto.

Acknowledgments

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-CONF-2019625). This material is based upon work supported by LLNL LDRD 25-ERD-042. This material was supported by funding from NSF (grant No. CCF-2028427). Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

References

[1] 2025. Teknium/OpenHermes-2.5 · Datasets at Hugging Face. https://huggingface.co/datasets/teknium/OpenHermes-2.5.
[2] 2026. Apache Arrow. https://arrow.apache.org/.
[3] 2026. Buffer Protocol. https://docs.python.org/3/c-api/buffer.html.
[4] 2026. Dask Scaling Limits. https://blog.dask.org/2018/06/26/dask-scaling-limits.
[5] 2026. Dask | Scale the Python Tools You Love. https://www.dask.org/.
[6] 2026. Deserialize Task Costs Too Much Time · Issue #3182 · Dask/Distributed. https://github.com/dask/distributed/issues/3182.
[7] 2026. Efficient IO with Io_uring. https://kernel.dk/io_uring.pdf.
[8] 2026. KiTS19 - Grand Challenge. https://kits19.grand-challenge.org/.
[9] 2026. Layouts :: Apache Log4j. https://logging.apache.org/log4j/2.x/manual/layouts.html.
[10] 2026. Llnl/Dfanalyzer. https://github.com/llnl/dfanalyzer.
[11] 2026. Llnl/Dftracer-Utils. https://github.com/llnl/dftracer-utils/tree/v0.0.5.
[12] 2026. Logs. https://opentelemetry.io/docs/concepts/signals/logs/.
[13] 2026. Lustre. https://www.lustre.org/.

[14] 2026. Module: Tf.Profiler | TensorFlow v2.16.1. https://www.tensorflow.org/api_docs/python/tf/profiler.

[15] 2026. Nanoarrow Documentation. https://arrow.apache.org/nanoarrow/latest/index.html.

[16] 2026. NVIDIA CUDA Profiling Tools Interface (CUPTI) - CUDA Toolkit. https://developer.nvidia.com/cupti.

[17] 2026. Pandas - Python Data Analysis Library. https://pandas.pydata.org/.

[18] 2026. Parquet. https://parquet.apache.org/.

[19] 2026. Parquet-Format/BloomFilter.Md at Master · Apache/Parquet-Format. https://github.com/apache/parquet-format/blob/master/BloomFilter.md.

[20] 2026. Perfetto - System Profiling, App Tracing and Trace Analysis. https://perfetto.dev/.

[21] 2026. Profiling Data of Scheduler.Update_graph for Very Large Graph · Issue #7998 · Dask/Distributed. https://github.com/dask/distributed/issues/7998.

[22] 2026. Protocol Buffers. https://protobuf.dev/.

[23] 2026. Python — Apache Arrow V24.0.0. https://arrow.apache.org/docs/python/index.html.

[24] 2026. [Python] PyArrow Cannot Read from a Newline-Delimited JSON File with Inconsistent Column Types, Even If Parse_options Specifies a Schema · Issue #49158 · Apache/Arrow. https://github.com/apache/arrow/issues/49158.

[25] 2026. Scaling to Large Datasets — Pandas 3.0.2 Documentation. https://pandas.pydata.org/docs/user_guide/scale.html.

[26] 2026. Shm_overview(7) - Linux Manual Page. https://man7.org/linux/man-pages/man7/shm_overview.7.html.

[27] 2026. SQLite Home Page. https://sqlite.org/.

[28] 2026. Torch.Profiler — PyTorch 2.10 Documentation. https://docs.pytorch.org/docs/stable/profiler.html.

[29] 2026. Trace Event Format. https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview?usp=embed_facebook.

[30] 2026. Using Rocprof — ROCProfiler 2.0.0 Documentation. https://rocm.docs.amd.com/projects/rocprofiler/en/latest/how-to/using-rocprof.html.

[31] 2026. Zlib/Examples/Zran.h at Develop · Madler/Zlib. https://github.com/madler/zlib/blob/develop/examples/zran.h.

[32] Jim Apple. 2023. Split Block Bloom Filters. arXiv:2101.01719 [cs] doi:10.48550/arXiv.2101.01719

[33] Eric Biggers. 2026. libdeflate: Heavily Optimized Library for DEFLATE/Zlib/Gzip Compression and Decompression. https://github.com/ebiggers/libdeflate

[34] Stanislav Bohm and Jakub Beranek. 2020. Runtime vs Scheduler: Analyzing Dask's Overheads. In *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. IEEE, GA, USA, 1–8. doi:10.1109/WORKS51914.2020.00006

[35] Peter Braam. 2019. The Lustre Storage Architecture. arXiv:1903.01955 [cs] doi:10.48550/arXiv.1903.01955

[36] Suren Byna, Jean Luca Bez, Tonglin Li, Quincey Koziol, Houjun Tang, and Qiao Kang. 2023. h5bench: A Benchmark Suite for Measuring HDF5 Performance. https://github.com/hpc-io/h5bench

[37] Özgün Çiçek, Ahmed Abdulkadir, Soeren S. Lienkamp, Thomas Brox, and Olaf Ronneberger. 2016. 3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation. arXiv:1606.06650 [cs] doi:10.48550/arXiv.1606.06650

[38] Darshan-HPC Project. 2026. Darshan: HPC I/O Characterization Tool. https://github.com/darshan-hpc/darshan

[39] Cameron Desrochers. 2026. moodycamel::ConcurrentQueue: A Fast Multi-Producer, Multi-Consumer Lock-Free Concurrent Queue for C++11. https://github.com/cameron314/concurrentqueue

[40] Hariharan Devarajan, Loïc Pottier, Kaushik Velusamy, Huihuo Zheng, Izzet Yildirim, Olga Kogiou, Weikuan Yu, Anthony Kougkas, Xian-He Sun, Jae Seung Yeom, and Kathryn Mohror. 2024. DFTracer: An Analysis-Friendly Data Flow Tracer for AI-Driven Workflows. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '24)*. IEEE Press, Atlanta, GA, USA, 1–24. doi:10.1109/SC41406.2024.00023

[41] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Trans. Storage* 17, 4 (Oct. 2021), 26:1–26:32. doi:10.1145/3483840

[42] Matt Godbolt. 2026. zindex: Create an Index on a Compressed Text File. https://github.com/mattgodbolt/zindex

[43] Google. 2026. Protocol Buffers: Google's Data Interchange Format. https://github.com/protocolbuffers/protobuf

[44] R. L. Graham. 1969. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.* 17, 2 (1969), 416–429. arXiv:https://doi.org/10.1137/0117039 doi:10.1137/0117039

[45] Nicholas Heller. 2026. KiTS19: The Official Repository of the 2019 Kidney and Kidney Tumor Segmentation Challenge. https://github.com/neheller/kits19

[46] Fabian Isensee, Philipp Kickingereder, Wolfgang Wick, Martin Bendszus, and Klaus H. Maier-Hein. 2019. No New-Net. arXiv:1809.10483 [cs] doi:10.48550/arXiv.1809.10483

[47] Maël Kerbiriou and Rayan Chikhi. 2019. Parallel Decompression of Gzip-Compressed Files and Random Access to DNA Sequences. arXiv:1905.07224 [cs] doi:10.48550/arXiv.1905.07224

[48] Maximilian Knespel and Holger Brunst. 2023. Rapidgzip: Parallel Decompression and Seeking in Gzip Files Using Cache Prefetching. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23)*. Association for Computing Machinery, New York, NY, USA, 295–307. doi:10.1145/3588195.3592992

[49] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*. Springer Berlin Heidelberg, 79–91. doi:10.1007/978-3-642-31476-6_7

[50] Daniel Lemire, Geoff Langdale, John Keiser, and simdjson contributors. 2026. simdjson: Parsing Gigabytes of JSON per Second. https://github.com/simdjson/simdjson

[51] Charles Masson, Jee E. Rim, and Homin K. Lee. 2019. DDSketch: A Fast and Fully-Mergeable Quantile Sketch with Relative-Error Guarantees. *Proceedings of the VLDB Endowment* 12, 12 (Aug. 2019), 2195–2205. doi:10.14778/3352063.3352135

[52] MLCommons. 2026. MLCommons Training: Reference Implementations of MLPerf ® Training Benchmarks. https://github.com/mlcommons/training

[53] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. 2009. ScalaTrace: Scalable Compression and Replay of Communication Traces for High-Performance Computing. *J. Parallel and Distrib. Comput.* 69, 8 (2009), 696–710. doi:10.1016/j.jpdc.2008.09.001

[54] Polars Project. 2026. Polars: A Fast Multi-Threaded Query Engine for DataFrames, Written in Rust. https://github.com/pola-rs/polars

[55] Felix Putze, Peter Sanders, and Johannes Singler. 2010. Cache-, Hash-, and Space-Efficient Bloom Filters. *ACM J. Exp. Algorithmics* 14 (Jan. 2010), 4:4.4–4:4.18. doi:10.1145/1498698.1594230

[56] Mark Raasveldt and Hannes Muehleisen. 2026. DuckDB: An Analytical In-Process SQL Database Management System. https://github.com/duckdb/duckdb

[57] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. arXiv:1910.02054 [cs] doi:10.48550/arXiv.1910.02054

[58] M. Rynge, G. Juve, J. Kinney, J. Good, B. Berriman, A. Merrihew, and E. Deelman. 2014. Producing an Infrared Multiwavelength Galactic Plane Atlas Using Montage, Pegasus, and Amazon Web Services. *Annual Astronomical Data Analysis Software and Systems (ADASS)* 485 (May 2014), 211.

[59] Frank Schmuck and Roger Haskin. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*. USENIX Association, USA, 16.

[60] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K. Lockwood, and Nicholas J. Wright. 2016. Modular HPC I/O Characterization with Darshan. In *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*. 9–17. doi:10.1109/ESPT.2016.006

[61] Houjun Tang, Quincey Koziol, John Ravi, and Suren Byna. 2022. Transparent Asynchronous Parallel I/O Using Background Threads . *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (April 2022), 891–902. doi:10.1109/TPDS.2021.3090322

[62] TensorFlow Developers. 2026. TensorBoard: TensorFlow's Visualization Toolkit. https://github.com/tensorflow/tensorboard

[63] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Roziè re, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs] doi:10.48550/arXiv.2302.13971

[64] GitHub User. 2026. DuckDB - Indexing. https://duckdb.org/docs/current/guides/performance/indexing.html.

[65] GitHub User. 2026. DuckDB - Loading JSON. https://duckdb.org/docs/current/data/json/loading_json.html.

[66] Chen Wang, Jinghan Sun, Marc Snir, Kathryn Mohror, and Elsa Gonsiorowski. 2020. Recorder 2.0: Efficient Parallel I/O Tracing and Analysis. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1–8. doi:10.1109/IPDPSW50202.2020.00176

[67] Xing Wu and Frank Mueller. 2013. Elastic and Scalable Tracing and Accurate Replay of Non-Deterministic Events. In *Proceedings of the 27th ACM International Conference on Supercomputing (ICS '13)*. doi:10.1145/2464996.2465001

[68] Cong Xu, Shane Snyder, Omkar Kulkarni, Vishwanath Venkatesan, Phillip Carns, Surendra Byna, Robert Sisneros, and Kalyana Chadalavada. 2019. *DXT: Darshan eXtended Tracing*. Technical Report. Lawrence Berkeley National Laboratory (LBNL), Berkeley, CA (United States).

[69] Izzet Yildirim, Hariharan Devarajan, Anthony Kougkas, Xian-He Sun, and Kathryn Mohror. 2025. WisIO: Automated I/O Bottleneck Detection with Multi-Perspective Views for HPC Workflows. doi:10.1145/3721145.3725742

[70] zlib-ng Project. 2026. zlib-ng: Zlib Data Compression Library for the Next Generation Systems. https://github.com/zlib-ng/zlib-ng