I/O Performance trade-offs among persistent layouts of RNTuple

S M Shovan

July 30, 2025

Fermilab National Laboratory

Outline

Context

- ROOT is a C++ framework developed at CERN for high-energy physics (HEP) data analysis and storage.
- Traditional storage uses TTree: columnar format for event-based data, but limited in scalability for exascale datasets.
- RNTuple is ROOT's next-generation columnar storage, offering improved compression, I/O performance, and flexibility.
- This project benchmarks RNTuple I/O configurations using simulated hit and wire data from particle detectors.
- Key data structures: Hits (particle interactions) and Wires (detector signals with ROIs).

Motivation

- HEP experiments generate petabytes of data; efficient I/O is critical for analysis workflows.
- RNTuple aims to outperform TTree in read/write speeds, file sizes, and multi-threaded access.
- Explore persistent layouts (e.g., vector vs. individual, split, spills) to optimize for real-world detector data patterns.
- Enable scalable parallel processing, reducing time-to-insight in large-scale simulations and experiments.

Problem Statement

- How do different RNTuple layouts affect I/O performance for hierarchical data like hits and wires?
- Evaluate trade-offs: file size, write times, cold/warm read times, and scaling with threads.
- Challenges: Balancing granularity for queries vs. overhead in entry counts; ensuring thread-safety in generation and I/O.
- Goal: Identify optimal configurations for HEP workloads, using benchmarks on generated datasets (e.g., 1M hits, 100K wires).

Objectives

- Benchmark RNTuple's I/O performance across diverse persistent layouts (vector, individual, split, spills) using simulated HEP data.
- Measure key metrics: write times, cold/warm read times, file sizes, and multi-threaded scaling.
- Analyze trade-offs between granularity, compression, and parallel efficiency for hierarchical datasets like hits and wires.
- Identify optimal configurations for high-throughput HEP workflows, enhancing ROOT's data handling capabilities.
- Provide actionable insights to guide RNTuple adoption in large-scale experiments.

Persistent Layouts

Data Layouts: AOS vs. SOA

Array of Structures (AOS)

Stores complete objects in an array. Good for row-wise access.

```
struct HitIndividual {
long long EventID;
unsigned int fChannel;
float fPeakTime;
};
// Array:  [Hit1, Hit2, ...]
```

Example: HitIndividual, WireIndividual (per-item entries).

Structure of Arrays (SOA)

Separate arrays per field. Optimized for columnar I/O.

```
struct HitVector {
vector<unsigned int> fChannel;
vector<float> fPeakTime;
};
// Columns:  fChannel[ ],
fPeakTime[ ], ...
```

Example: HitVector, WireVector (per-event vectors).

Project Use: AOS for granular queries; SOA for batch efficiency in RNTuple storage.

ROI Flattening vs. Custom Dictionary

ROI Flattening (Non-Dictionary)

Flattens hierarchical ROI data into vectors for efficient storage without custom classes.

```
struct WireVector {
vector<unsigned int>
fSignalROI_nROIs;
vector<size_t> fSignalROI_offsets;
vector<float> fSignalROI_data;
};
```

Example: Used in non-dictionary experiments for raw vector-based I/O.

Custom Dictionary (ROOT Classes)

Uses structured classes with ClassDef for ROOT's dictionary system, enabling object-oriented I/O.

```
struct RegionOfInterest {
size_t offset;
vector<float> data;
ClassDef(RegionOfInterest, 3)
};
```

Example: Used in dictionary experiments for type-safe, hierarchical data handling.

Project Use: Flattening optimizes for speed; dictionaries provide better data integrity and queryability.

Layout Strategies

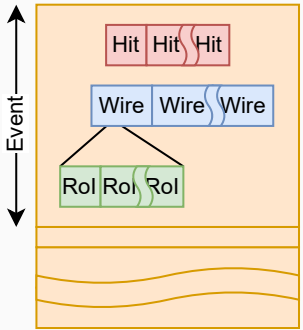


Figure 1: AOS Layout

Layout Strategies

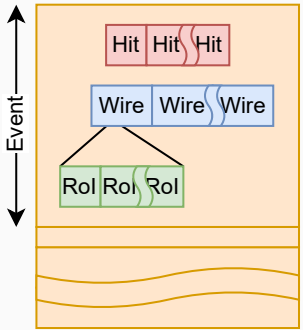


Figure 1: AOS Layout

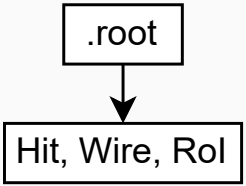


Figure 2: Vertical Layout 1

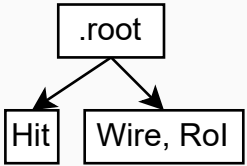


Figure 3: Vertical Layout 2

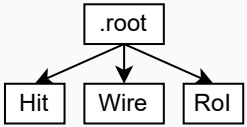


Figure 4: Vertical Layout 3

Layout Strategies

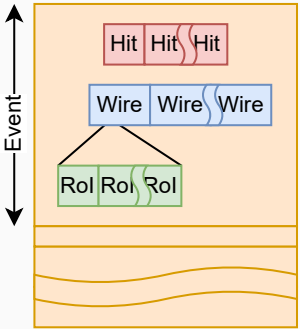


Figure 1: AOS Layout

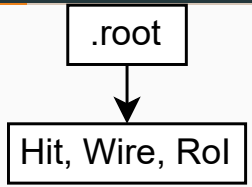


Figure 2: Vertical Layout 1

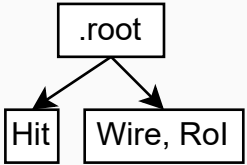


Figure 3: Vertical Layout 2

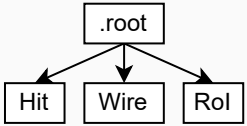


Figure 4: Vertical Layout 3

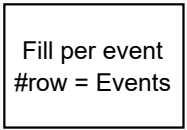


Figure 5: Horizontal Layout 1

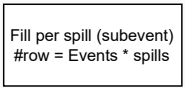


Figure 6: Horizontal Layout 2

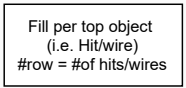


Figure 7: Horizontal Layout 3

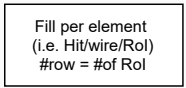
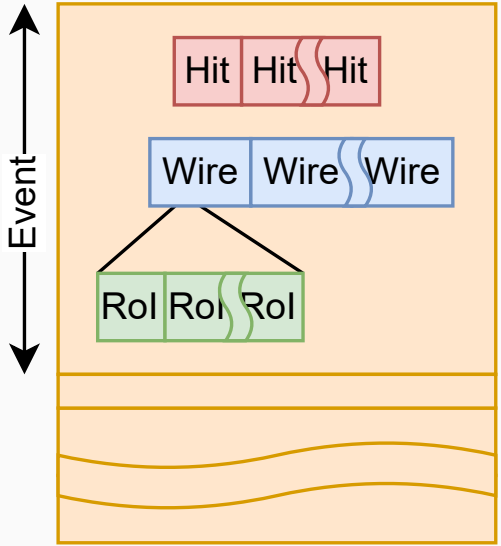
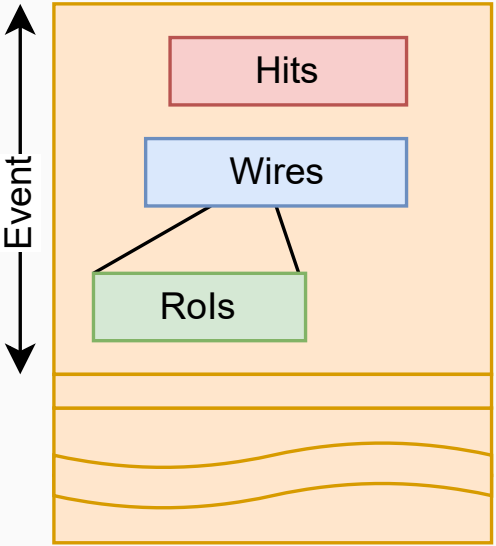


Figure 8: Horizontal Layout 4

Layout Strategies: AOS vs SOA





Memory/Data Considerations

Memory/Data Considerations

Parallel Optimizations

Write Optimization: Multi-Threaded Chunking

Parallel Chunking Divide events into thread-specific ranges for concurrent generation and filling.

```
std::vector<unsigned int> seeds =
generateSeeds(nThreads);
for (int th = 0; th < nThreads;
++th) {
int first = th * chunkSize;
int last = std::min(first +
chunkSize, totalEvents);
futures.emplace_back(std::async(
std::launch::async, thinWorkFunc,
first, last, seeds[th], th
));
}
```

Example: From `executeInParallel` in

Synchronized Filling Use mutex to safely flush clusters after filling.

```
{ std::lock_guard<std::mutex>
lock(mutex); }
hitContext.FlushCluster();
```

Project Use: Scales writes with cores for large datasets.

Read Optimization: Cluster-Aware Splitting

Cluster Splitting Split read ranges by cluster boundaries to avoid duplicates.

```
auto clusters =
split_range_by_clusters(*reader,
nChunks);
for (auto& chunk :  clusters) {
futures.push_back(std::async(
&processChunk, chunk.first,
chunk.second
));
}
```

Example: From read functions in `HitWireReaders.cpp`.

Benefits Prevents redundant reads, optimizes multi-threaded access.

```
std::vector<std::pair<size_t,
size_t>>
split_range_by_clusters(ROOT::RNTupleReade
reader, int nChunks)
```

Project Use: Enhances warm read efficiency.

Parallel Write Challenge: File Corruption Solution

Problem: Concurrent Flushes

Unsynchronized cluster flushes cause file corruption in multi-threaded writes. **Example**

Issue: Threads overwriting shared file regions.

Solution: Mutex Synchronization Lock during flushes to serialize access per cluster.

```
for (int idx = first; idx < last;
++idx) {
// Generate data for hits/wires
if (hitStatus.ShouldFlushCluster())
{
hitContext.FlushColumns();
{ std::lock_guard<std::mutex>
lock(mutex); }
hitContext.FlushCluster();
}
}
```

Project Use: Ensures thread-safe parallel writes without corruption.

Limitations

Results

Evaluation Metrics

Performance Analysis

Experiment 1

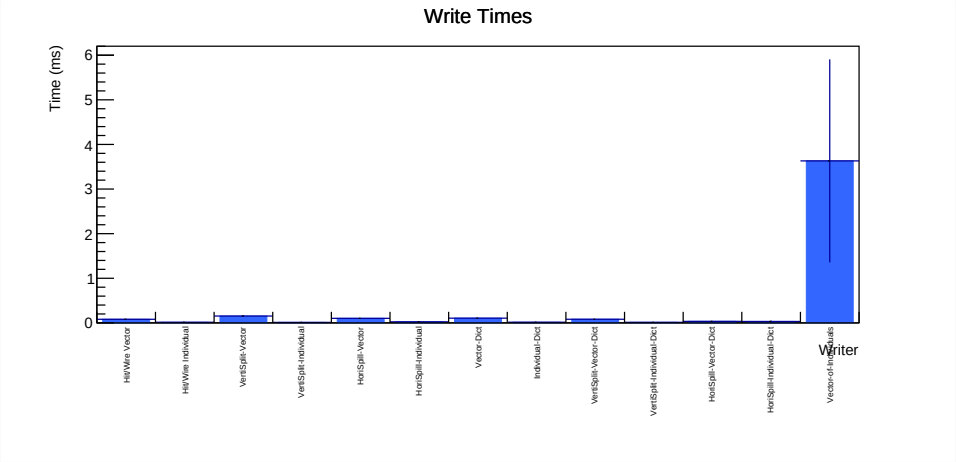


Figure 11: Experiment 1: Description goes here.

Experiment 2

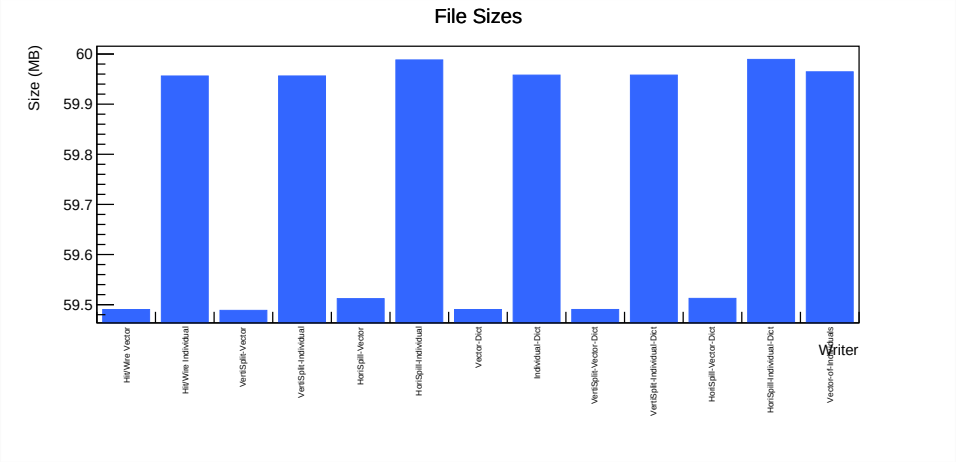


Figure 12: Experiment 2: Description goes here.

Experiment 3

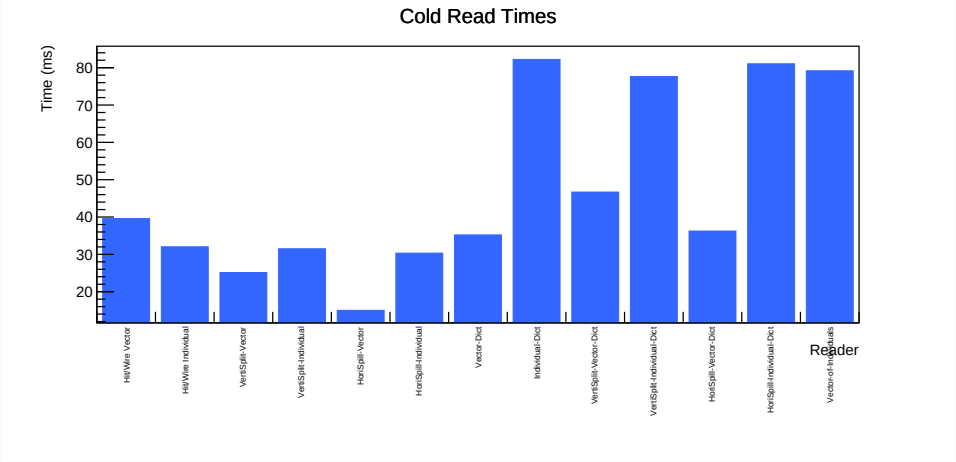


Figure 13: Experiment 3: Description goes here.

Experiment 4

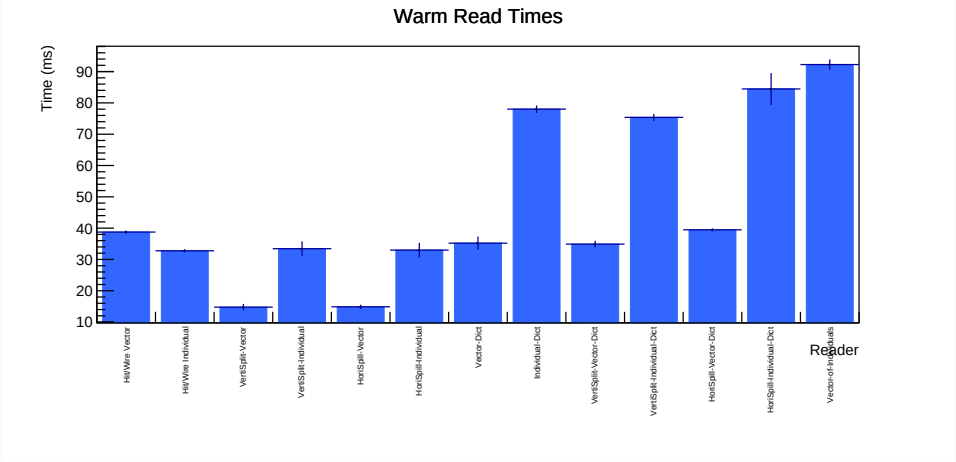
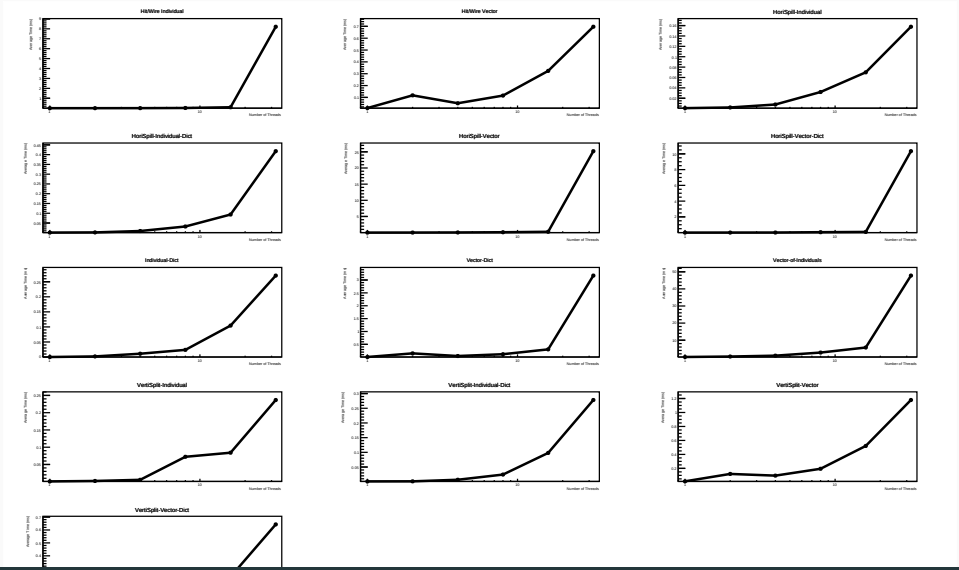


Figure 14: Experiment 4: Description goes here.

Experiment 5



Trade-offs

Design Trade-offs

Future Considerations

Thank you!
Questions?