I/O Performance trade-offs among RNTuple's persistent layouts for DUNE Data Products

S M Shovan, *FCSI Summer Intern'25*

July 31, 2025

Fermi National Accelerator Laboratory

Outline

Introduction

Persistent Layouts

Parallel Optimizations

Challenges

Results

Trade-offs

Introduction

Introduction

- The Deep Underground Neutrino Experiment (DUNE) is projected to record roughly 30 PB of liquid-argon TPC data per year [1]—far beyond the scale of previous neutrino experiments.
- DUNE's **Phlex** test-stand at Fermilab provides realistic hit/wire samples to prototype read-out and offline storage.
- ROOT's new `RNTuple` backend is a candidate for the long-term DUNE data model, promising faster compression, cluster-aware reads, and thread-safe writes.
- This study benchmarks alternative RNTuple *persistent layouts* (AOS/SOA, vertical splits, granularity levels) using Phlex-generated Hit and Wire data products.
- Focus data products: `recob::Hit` (charge deposits) and `recob::Wire` (ROI-compressed waveforms).

Motivation

- A single DUNE far-detector module streams about 1.2 TB/s of raw data before compression [2]; naive storage could potentially overwhelm the archival budget.
- Efficient layout choice can cut file size and accelerate cluster reads needed for GPU/CPU reconstruction farms.

Problem Statement

- Which RNTuple persistent layout minimises *both* write time and on-disk footprint for DUNE Hit/Wire hierarchies?
- How does vertical splitting (all-DP, per-DP, per-group) interact with horizontal granularities (event, spill, element) under Phlex workloads?

Objectives

- Benchmark seven layout variants on a 1 M-event Phlex dataset (`recob::Hit`, `recob::Wire` with ROIs).
- Measure: write throughput, cold/warm read latency, compressed file size, and multi-thread scaling (1–64 threads).
- Quantify trade-offs of ROI flattening, vertical split depth, and SOA vs. AOS in the context of DUNE's access patterns.

Persistent Layouts

Data Layouts: AOS vs. SOA

Array of Structures (AOS)

Stores complete objects in an array.

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

Example: Hit, Wire (per-item entries).

Structure of Arrays (SOA)

Separate arrays per field.

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

Example: Hits, Wires (per-event vectors).

Layout Strategies

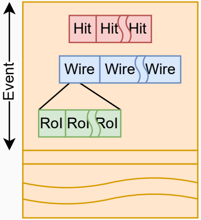

Figure 1: AOS Layout

Layout Strategies

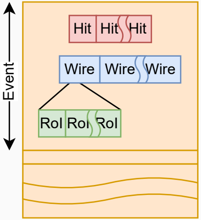

Figure 1: AOS Layout

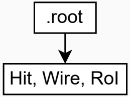

Figure 2: 1 RNTuple for all data products

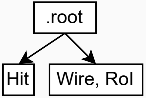

Figure 3: 1 RNTuple per data product

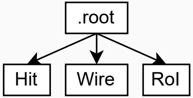

Figure 4: 1 RNTuple per group

Layout Strategies

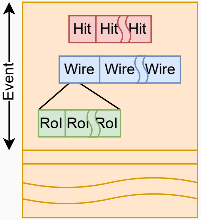

Figure 1: AOS Layout

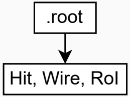

Figure 2: 1 RNTuple for all data products

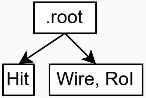

Figure 3: 1 RNTuple per data product

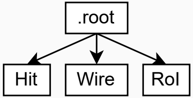

Figure 4: 1 RNTuple per group

Fill per event
#row = Events

Figure 5: 1 fill/row per event

Fill per spill (subevent)
#row = Events * spills

Figure 6: 1 fill/row per spill

Fill per top object
(i.e. Hit/wire)
#row = #of hits/wires

Figure 7: 1 fill/row per top object

Fill per element
(i.e. Hit/wire/RoI)
#row = #of RoI

Figure 8: 1 fill/row per element

Layout Strategies: AOS vs SOA

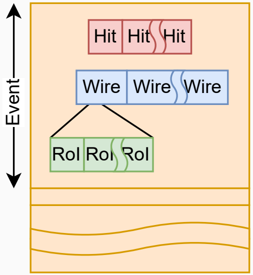

Figure 9: AOS Layout

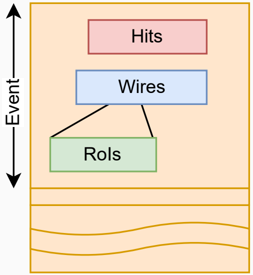

Figure 10: SOA Layout

Granularity vs. Vertical Split Matrix

Horizontal Granularity	1 NTuple (all DP)	1 NTuple / DP	1 NTuple / group
Event-wise	`event_allDP()`	`event_perDP()`	`event_perGroup()`
Spill-wise	`spill_allDP()`	`spill_perDP()`	`spill_perGroup()`
Top-object-wise	–	`topObject_perDP()`	`topObject_perGroup()`
Element-wise	–	`element_perDP()`	`element_perGroup()`

DP = Data Product

Memory/Data Considerations

Memory/Data Considerations

Parallel Optimizations

Write Optimization: Multi-Threaded Chunking

Parallel Chunking Divide events into thread-specific ranges for concurrent 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: `executeInParallel` writers.

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
));
}
```

Benefits Prevents redundant reads, optimizes multi-threaded access.

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

Project Use: Enhances read efficiency.

Challenges

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.

ROI Flattening vs. Custom Dictionary

ROI Flattening (Non-Dictionary)

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

```
struct Wires {
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.

Results

Evaluation Metrics

- **Write Throughput:** Total events per second during RNTuple serialization.
- **Cold Read Time:** Latency for first access after file creation, reflecting raw I/O.
- **Warm Read Time:** Latency when data is cached, measuring memory locality effects.
- **Compressed File Size:** Total on-disk footprint post-write, accounting for RNTuple's column-wise compression.
- **Multi-thread Scaling:** Performance variation from 1 to 64 threads for write and read workloads.

AOS vs SOA: Write Performance Comparison

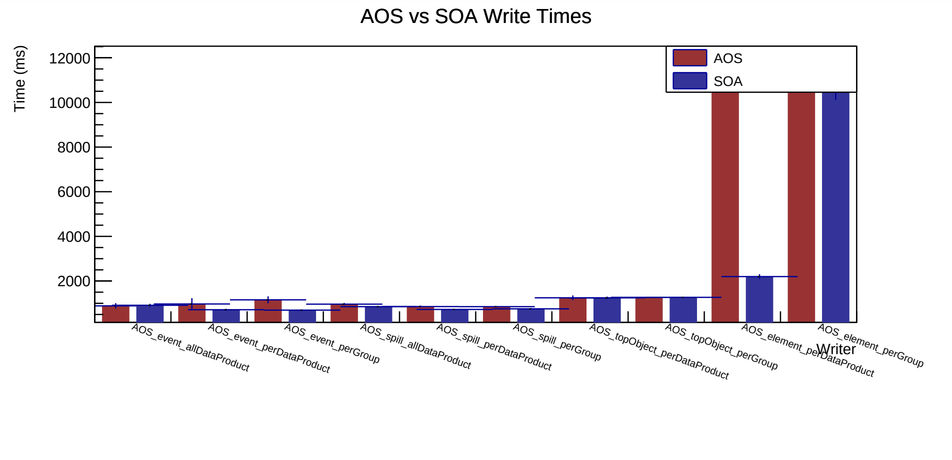

Figure 11: Write Performance Comparison: AOS vs SOA approaches across different data organization strategies. Red bars represent AOS (Array of Structures) performance, while blue bars show SOA (Structure of Arrays) performance

CPU Time Overheads

Writer	A	B	C
`event_perGroup`	✓	✓	
`spill_perGroup`	✓	✓	✓
`element_perData`	✓	✓	
`topObject_perData`		✓	
`event_perData`		✓	
`spill_perData`		✓	✓

- **A – ROI flattening:** dominant for *element* and *perGroup* writers.
- **B – Multiple NTuples:** extra `Fill`/Flush cycles & mutex contention.
- **C – Spill re-partitioning:** additional work unique to *spill* writers.

AOS vs SOA: Cold Read Performance Comparison

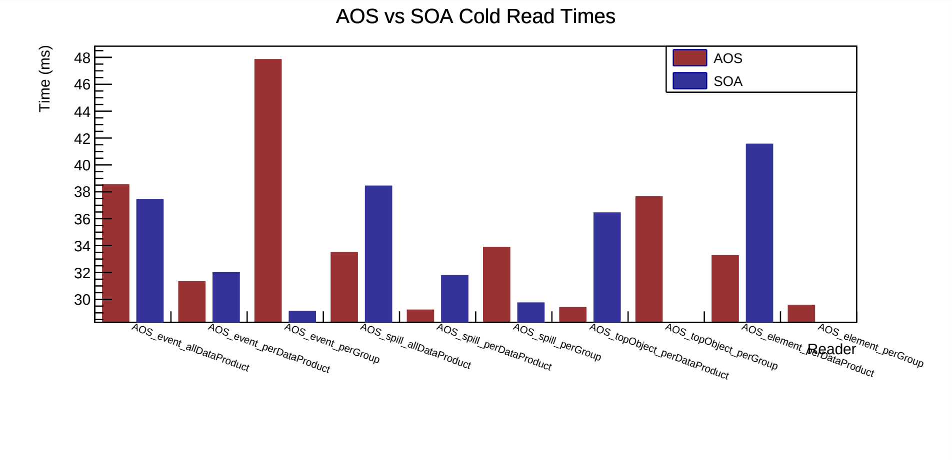

Figure 12: Cold Read Performance Comparison: Initial read times for AOS vs SOA implementations. Cold reads represent the first access to data after file creation, measuring baseline I/O performance

AOS vs SOA: Warm Read Performance Comparison

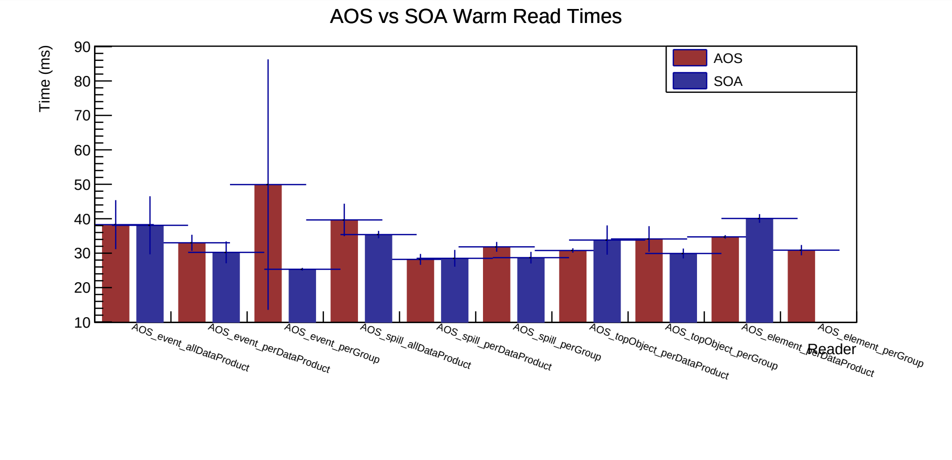

Figure 13: Warm Read Performance Comparison: Subsequent read times for AOS vs SOA implementations. Warm reads benefit from cached data, showing optimized access patterns and memory locality effects.

AOS vs SOA: File Size Comparison

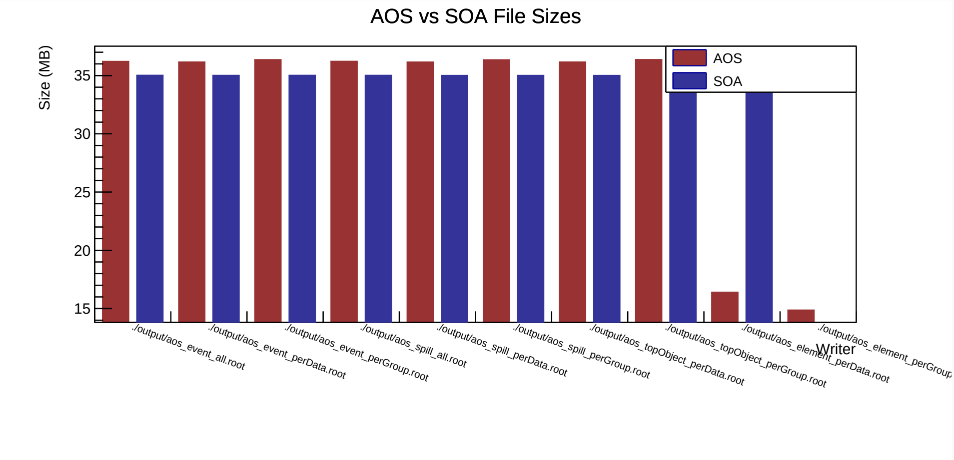

Figure 14: File Size Comparison: Storage efficiency of AOS vs SOA approaches. File sizes impact storage costs, network transfer times, and overall data management overhead in large scale HEP experiments

File Size Insights

Writer variant	Size (% of most)
`aos_element_perData.root`	47%
`soa_element_perData.root`	100%
`aos_element_perGroup.root`	42%
`soa_element_perGroup.root`	39%
`aos_event_* / aos_spill_*`	~103%
`soa_event_* / soa_spill_*`	100%

Key Take-aways

- Deep nesting & high row counts double basket overhead (offset tables, headers, CRC).
- Flattened `perGroup` writers cut size roughly in half by removing ROI nesting.

Trade-offs

Trade-off Analysis

Future Considerations

References i

[1] DUNE Collaboration, “Deep Underground Neutrino Experiment Technical Design Report–VolumeII: DUNE Physics,” 2020, Sec.2.6.

[2] DUNE Collaboration, “Data Acquisition System for the DUNE Far Detector,” IEEE NSS/MIC Proc., 2023.

Thank you!

Questions?