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

S M Shovan, *FCSI Summer Intern'25*

August 12, 2025

Fermi National Accelerator Laboratory

Outline

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 is developing a new framework: **Phlex** stands for **P**arallel, **h**ierarchical, and **l**ayered **ex**ecution of data-processing algorithms.
- ROOT's new `RNTuple` storage container 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) for realistic 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 read time, write time and on-disk footprint for DUNE Hit/Wire data products?
- How does vertical splitting such as one RNTuple for all data products or one of each data products interact with horizontal granularities (event, spill, element)?
- How does the choice of persistent layout affect the performance of the read and write operations?

Objectives

- Benchmark seven layout variants on a 1 M-event (35 GB) Phlex dataset (`recob::Hit`, `recob::Wire` with ROIs).
- Measure: write throughput, cold/warm read latency, compressed file size.
- Quantify trade-offs of ROI flattening, vertical split depth, and SOA vs. AOS.

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<long long> EventID;
   vector<unsigned int> fChannel;
   vector<float> fPeakTime;
};
// Columns:  EventID[ ],
//fChannel[ ], fPeakTime[ ], ...
```

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

Layout Strategies

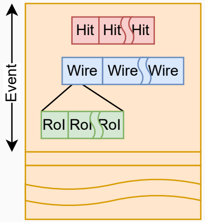


Figure 1: AOS Layout

Layout Strategies

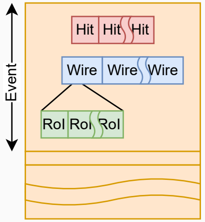


Figure 1: AOS Layout

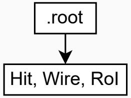


Figure 2: 1 RNTuple for all data products

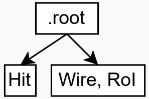


Figure 3: 1 RNTuple per data product

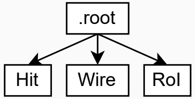


Figure 4: 1 RNTuple per group

Layout Strategies

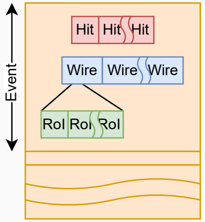


Figure 1: AOS Layout

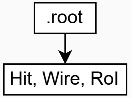


Figure 2: 1 RNTuple for all data products

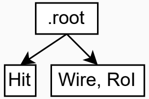


Figure 3: 1 RNTuple per data product

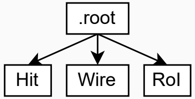


Figure 4: 1 RNTuple per group

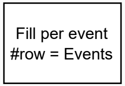


Figure 5: 1 fill/row per event

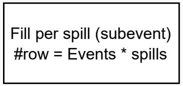


Figure 6: 1 fill/row per spill

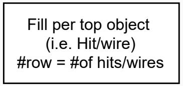


Figure 7: 1 fill/row per top object

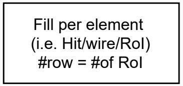


Figure 8: 1 fill/row per element

Layout Strategies: AOS vs SOA

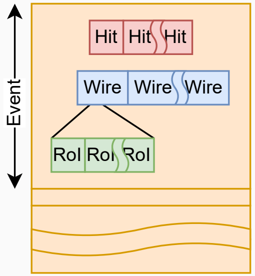


Figure 9: AOS Layout

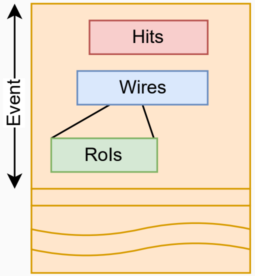


Figure 10: SOA Layout

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

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.

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
    ));
}
```

Helper Function Defines cluster-based splits.

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

Project Use: Enhances read efficiency by reducing redundant reads.

Challenges

Challenge: Corrupted ROOT Files

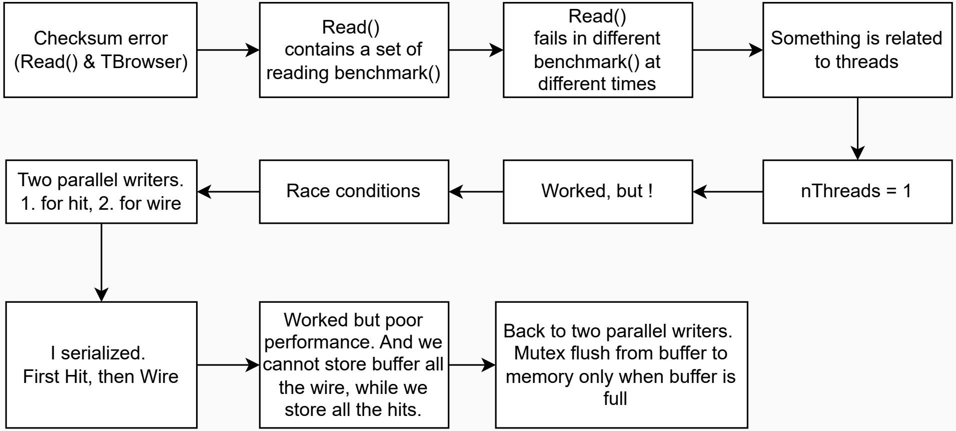

Checksum error (Read() & TBrowser)
Read() contains a set of reading benchmark()
Read() fails in different benchmark() at different times
Something is related to threads
nThreads = 1
Worked, but !
Race conditions
Two parallel writers. 1. for hit, 2. for wire
I serialized. First Hit, then Wire
Worked but poor performance. And we cannot store buffer all the wire, while we store all the hits.
Back to two parallel writers. Mutex flush from buffer to memory only when buffer is full

Figure 11: Challenge: Addressing corrupted ROOT files in parallel write operations.

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 of ROOT's dictionary system, enabling object-oriented I/O.

```
struct RegionOfInterest {
    size_t offset;
    vector<float> data;
};
```

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.

AOS: Write Performance

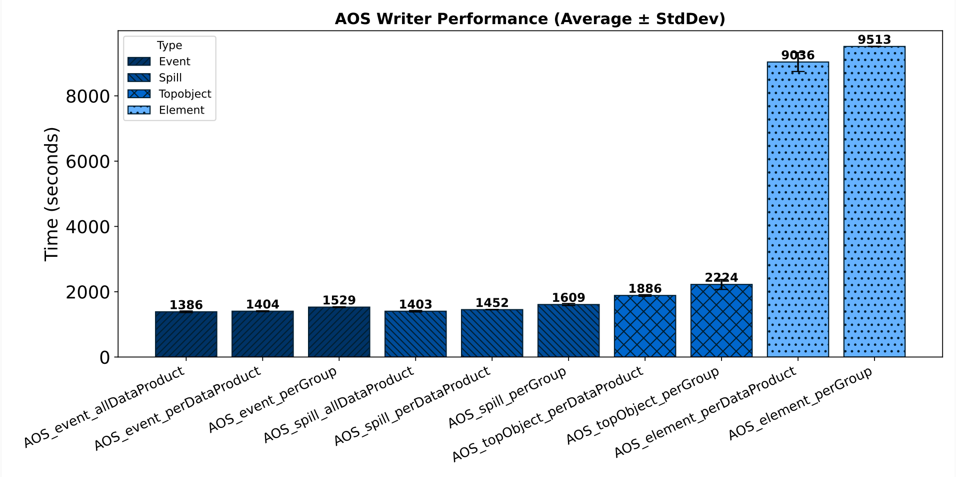
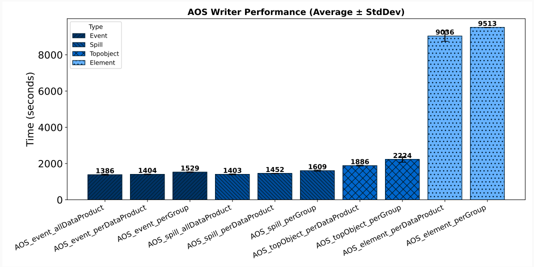


Figure 12: AOS (Array of Structures) Write performance across different persistent layouts.

AOS: Write Performance



Key Takeaways:

- Higher granularity leads to slower write performance due to thread contention.
- For horizontal persistent layouts, slow down is upto 6.9×, although it is marginal for vertical persistent layouts.

SOA: Write Performance

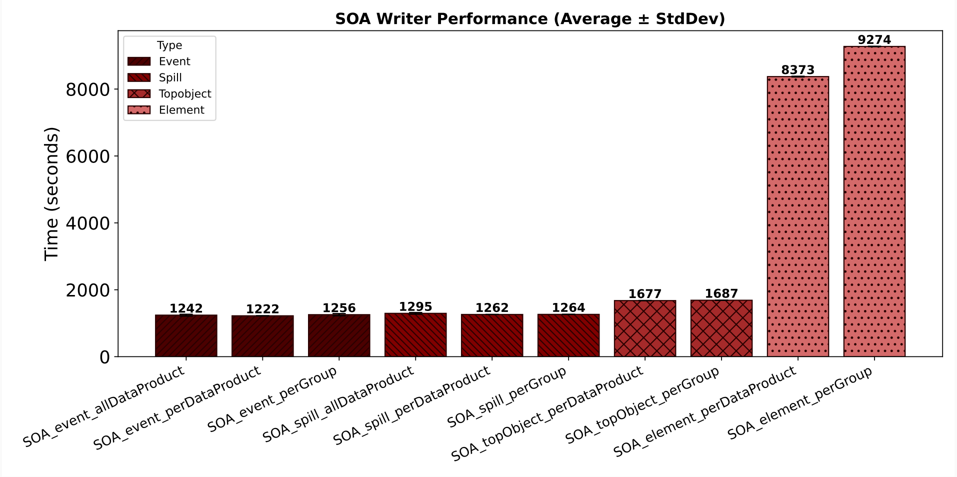
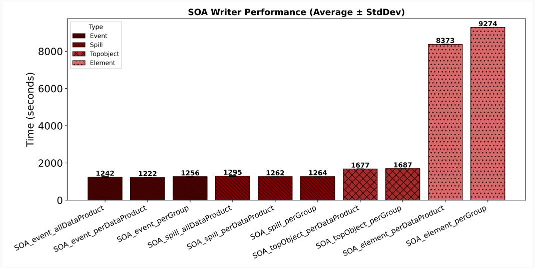


Figure 13: SOA (Structure of Arrays) Write performance across different persistent layouts.

SOA: Write Performance



Key Takeaway:

- SOA writer performance is overall similar to AOS writer performance.

AOS vs SOA: Write Performance

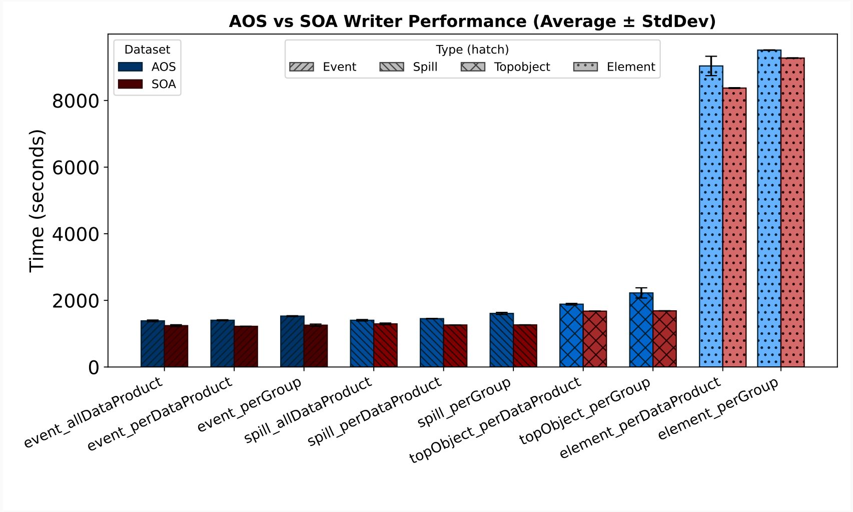
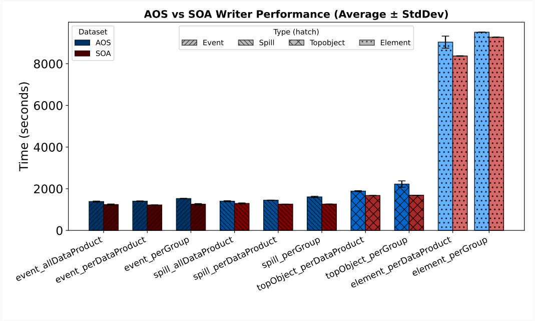


Figure 14: AOS vs SOA write performance across different persistent layouts.

AOS vs SOA: Write Performance



Key Takeaway:

- SOA writer is on average 3.65% faster than AOS writer for all persistent layouts.

File Size Analysis: AOS vs SOA

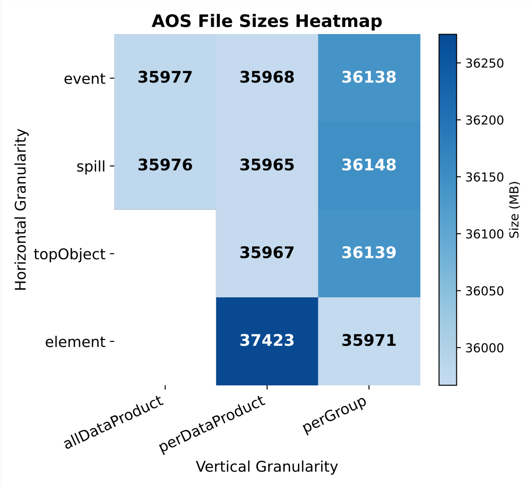


Figure 15: AOS file size across persistent layouts

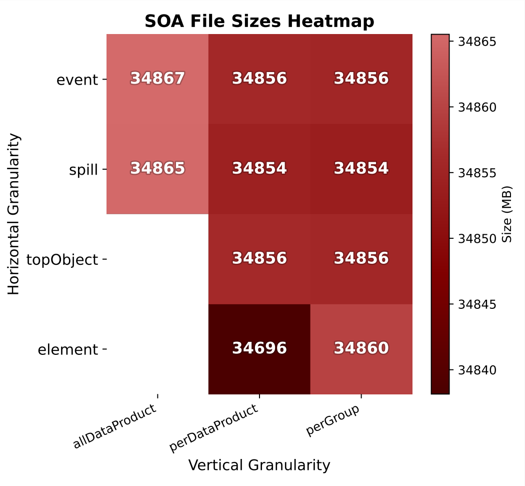
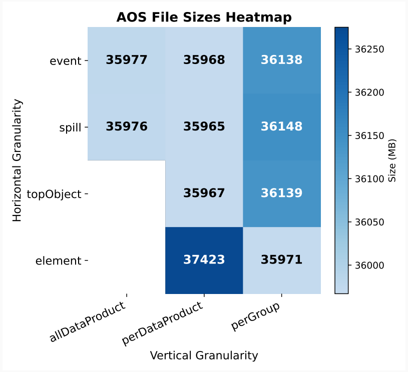
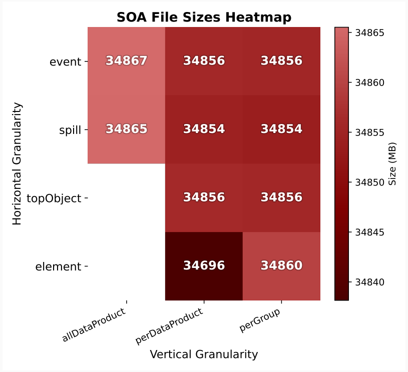


Figure 16: SOA file size across persistent layouts

File Size Analysis: AOS vs SOA





Inconclusive observations:

- The variability in file size is higher for AOS than SOA.
- Element_perDataProduct layout for AOS leads to higher file size due to additional information storage of EventID and WireID.

Fields Read in Benchmarks

- **Hits**: `PeakAmplitude`.
- **Wires**: `Channel`.
- **ROIs**: ROI `data vector` `data`.

- **Note**: We do not reconstruct full objects from the fields; we only read the fields through RNTuple views, using `volatile` to prevent the compiler optimization.

AOS vs SOA: Cold Read Performance Comparison

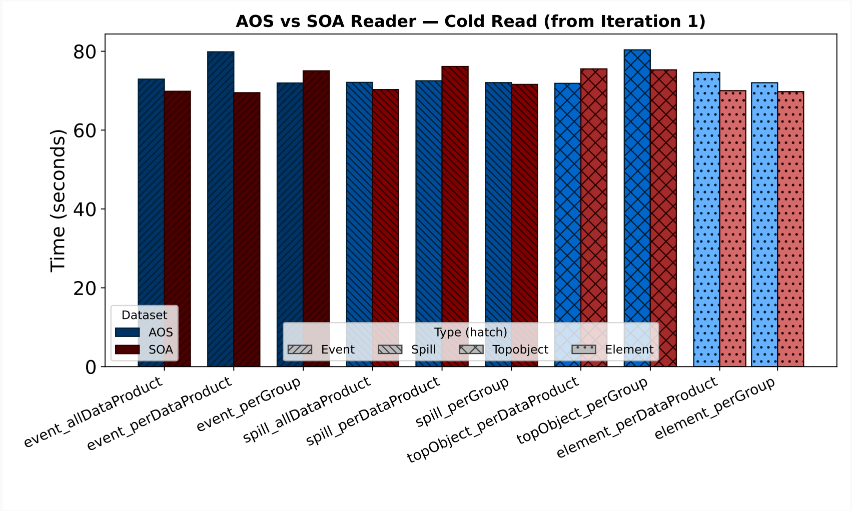
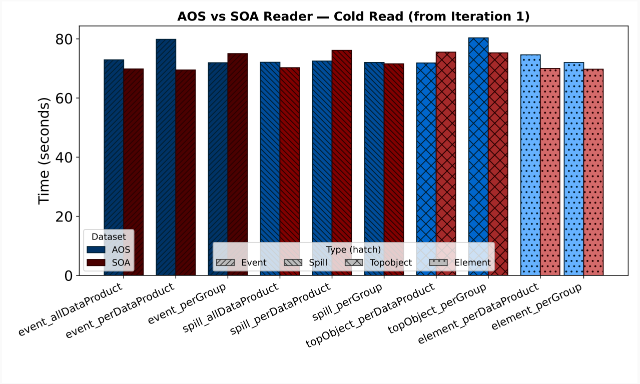


Figure 17: Cold Read Performance Comparison: Initial read times for AOS vs SOA implementations.

AOS vs SOA: Cold Read Performance Comparison



Inconclusive observations:

- SOA reader is on average 2.34% faster than AOS reader for cold read.

AOS vs SOA: Warm Read Performance Comparison

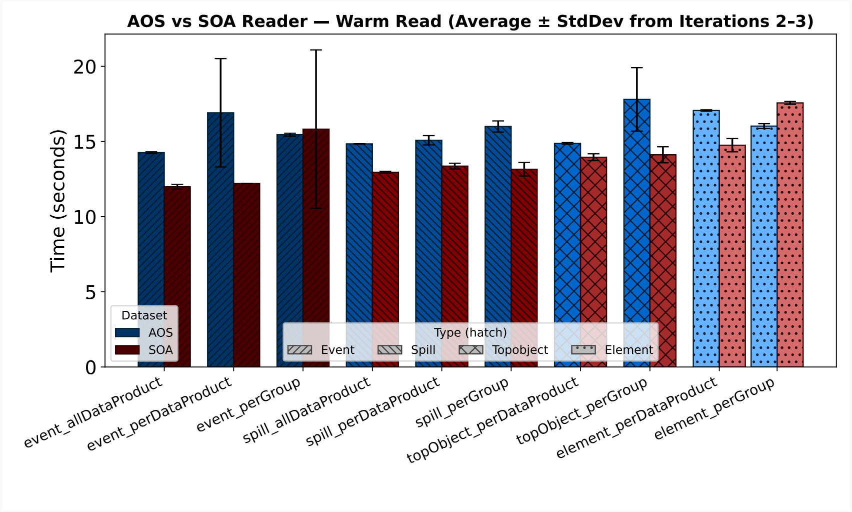
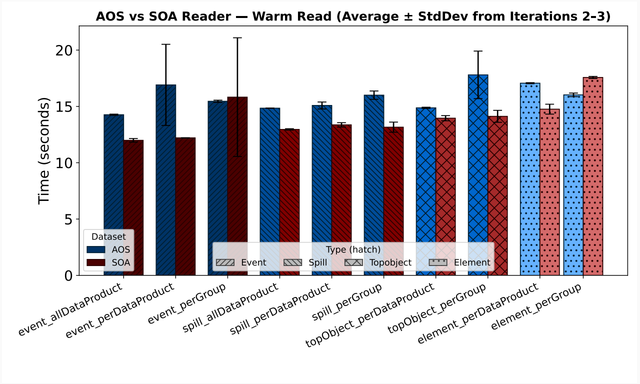


Figure 18: Warm Read Performance Comparison: Subsequent read times for AOS vs SOA implementations.

AOS vs SOA: Warm Read Performance Comparison



Inconclusive observations:

- Average warm read is 4.91× faster than cold read across all persistent layouts.
- SOA reader is on average 11.63% faster than AOS reader for warm read.

Future Considerations

- **Uniformity in Data Storage:** Deterministic approach to store exactly same data for each of the root files.
- **Thread Scaling Analysis:** Rigorous testing across 1–128 threads to evaluate layout performance scaling and optimal configurations.
- **Extensible Framework:** Develop scalable architecture for arbitrary data products beyond Hits and Wires with configurable layouts.
- **Advanced Layout Testing:** Explore N-tuple groupings and clustering strategies for improved read/write efficiency and storage optimization.

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?