Evolution of the ROOT Tree I/O

Jakob Blomer[1], Philippe Canal[2], Axel Naumann[1], Danilo Piparo[1]
[1]CERN [2]Fermilab

CHEP 2019

The TTree I/O

TTree's column-wise format addresses our very problem

- TTree I/O speed and storage efficiency is significantly better than industry products ▸ ACAT'17
- Only few other formats can serialize the complexity of even the simplest event models:
  - Apache Parquet: optimized for sparse collections but HEP data is not sparse
  - Apache Arrow: only in-memory format but not on-disk format
- ROOT's unique feature: **seamless C++ integration**, users do not need to write or generate schema mapping

Nested collections typical in HEP

```
struct Event {
  std::vector<Particle> fPtcls;
};

struct Particle {
  std::vector<Track> fTracks;
};

struct Track {
  int fVertexId;
};
```

We want to ensure that ROOT I/O continues to yield the most efficient analysis I/O

The TTree I/O

TTree's column-wise format addresses our very problem

- TTree I/O speed and storage efficiency is significantly better than industry products ▸ ACAT'17
- Only few other formats can serialize the complexity of even the simplest event models:
  - Apache Parquet: optimized for sparse collections but HEP data is not sparse
  - Apache Arrow: only in-memory format but not on-disk format
- ROOT's unique feature: **seamless C++ integration**, users do not need to write or generate schema mapping

Nested collections typical in HEP

```
struct Event {
  std::vector<Particle> fPtcls;
};

struct Particle {
  std::vector<Track> fTracks;
};

struct Track {
  int fVertexId;
};
```

We want to ensure that ROOT I/O continues to yield the most efficient analysis I/O

RNTuple: The New Experimental ROOT I/O Subsystem

Event iteration
`RNTupleDataSource (RDF)`, `RNTupleView`, `RNTupleReader/Writer`

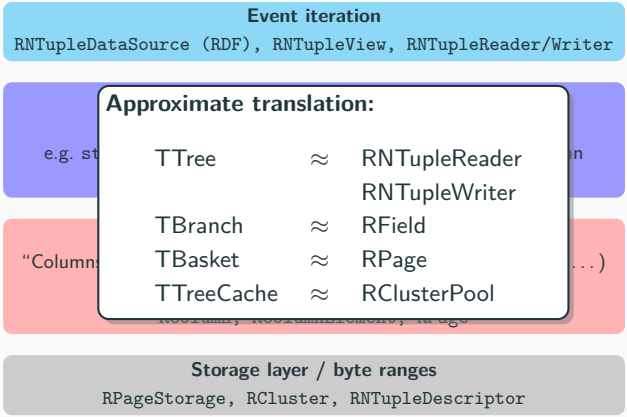
Logical layer / C++ objects
Mapping of C++ types onto columns
e.g. `std::vector<float>` $\mapsto$ index column and a value column
`RField`, `RNTupleModel`, `REntry`

Primitives layer / simple types
"Columns" containing elements of fundamental types (float, int, ...)
grouped into (compressed) pages and clusters
`RColumn`, `RColumnElement`, `RPage`

Storage layer / byte ranges
`RPageStorage`, `RCluster`, `RNTupleDescriptor`

- Improve different parts independently
- Add new storage backends
  - Physical: ROOT file container, raw file, object store, NVRAM
  - Virtual: "friend" and "chain"
- Serialization of simple types and STL collections built-in – can be read without libCore

RNTuple: The New Experimental ROOT I/O Subsystem



- Improve different parts independently
- Add new storage backends
  - Physical: ROOT file container, raw file, object store, NVRAM
  - Virtual: "friend" and "chain"
- Serialization of simple types and STL collections built-in – can be read without libCore

RNTuple format evolution

Key improvements in RNTuple

- More efficient storage of collections and sub collections
- On disk layout matches most modern in-memory layouts (little-endian)
- Better control of I/O memory utilization
- Boolean values stored as bit field instead of byte array

Goal: slash memory copies and (virtual) function calls in I/O code paths

RNTuple's type system

- `bool`
- Integers (signed and unsigned, 8bit to 64bit)
- `float`, `double`
- `std::string`
- `std::array`
- `std::vector`, `VecOps::RVec`
- `std::variant`
- Whatever other `std` type we want, e. g. `std::chrono`
- Classes with dictionaries

Fully composable within the type system

RNTuple Interface Sketch: Reading Data

Populate User Objects

```
auto model = RNTupleModel::Create();
auto fldPt = model->MakeField<float>("pt");
// Note: there is also a type-unsafe API

auto ntpl = RNTupleReader::Open(
   std::move(model), "Events", "f.root");

for (auto entryId : *ntpl) {
   ntuple->LoadEntry(entryId);
   h.Fill(*fldPt);
}
```

Zero-Copy

```
auto ntpl =
   RNTupleReader::Open("Events", "f.root");
auto viewPt = ntpl->GetView<float>("pt");

for (auto i : ntpl->GetViewRange()) {
   h.Fill(viewPt(i));
}
```

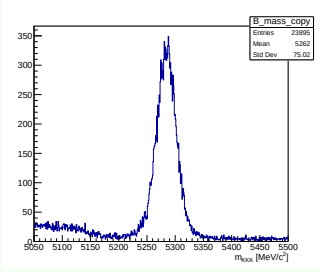
RDataFrame

```
auto df = ROOT::Experimental::MakeNTupleDataFrame("Events", "http://xrootd/f.root");
```

Sample Analyses for Performance Benchmarking

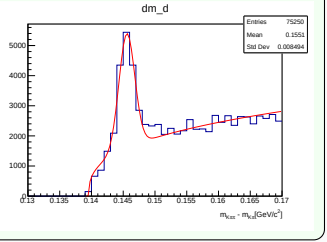
LHCb run 1 open data B2HHH

- Dense reading ($> 75\,\%$): 18/26 branches
- Fully flat data model
- 8.5 million events
- 24 k selected events



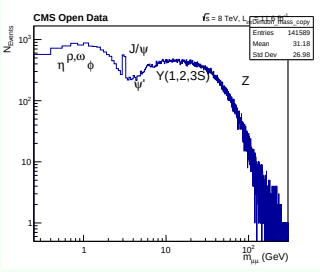
H1 micro dst [$\times 10$]

- Medium dense reading ($\sim 10\,\%$): 16/152 branches
- Event substructure: vector of jets etc.
- 2.8 million events
- 75 k selected events

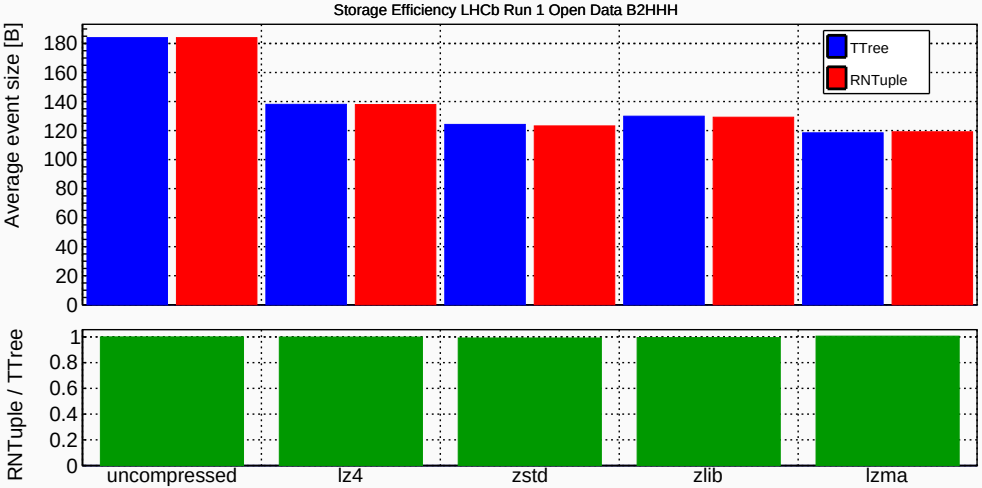


CMS nanoAOD June 2019

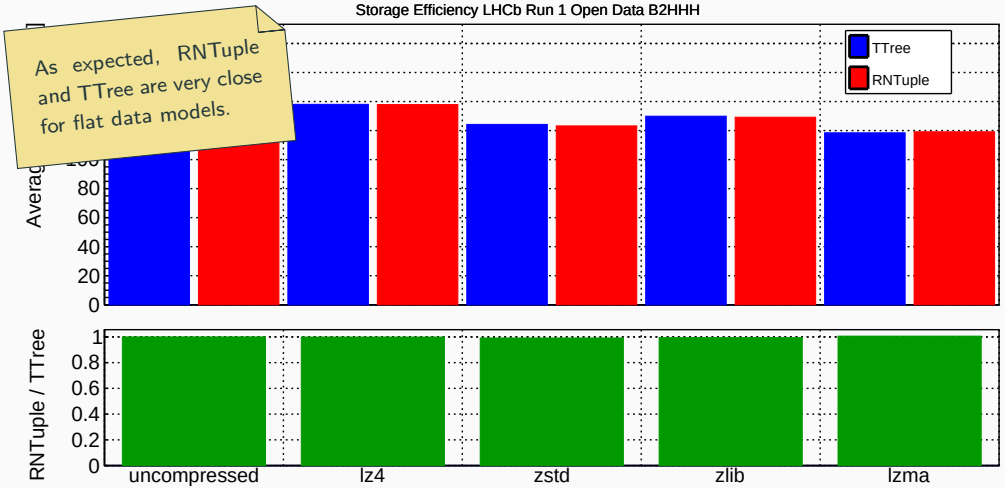
- Sparse reading ($< 1\,\%$): 6/1479 branches
- Event substructure: vector of jets etc.
- 1.6 million events
- 141 k selected events



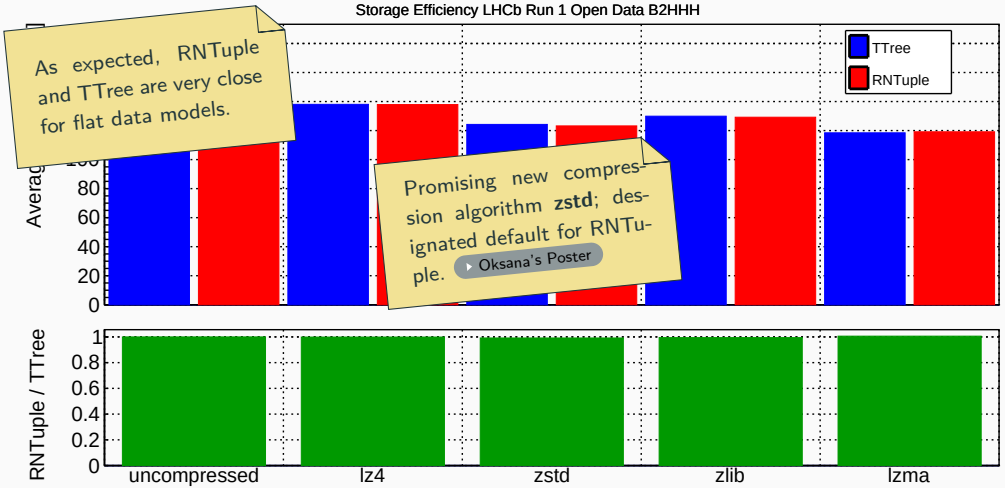
Storage efficiency for very simple data models



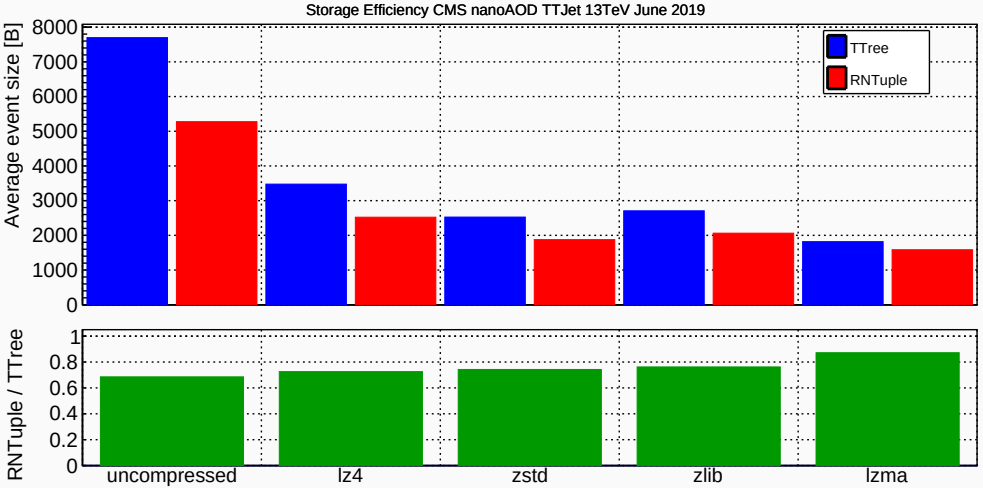
Storage efficiency for very simple data models



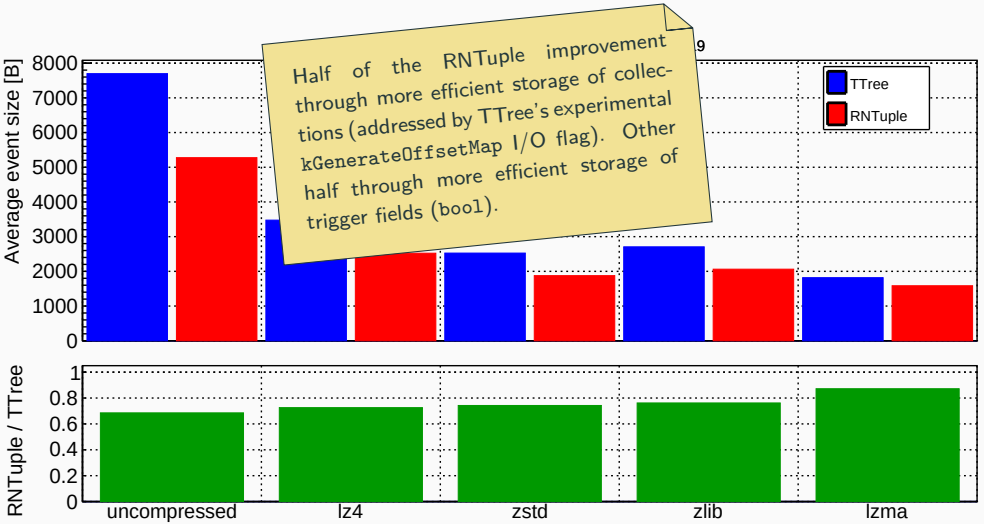
Storage efficiency for very simple data models



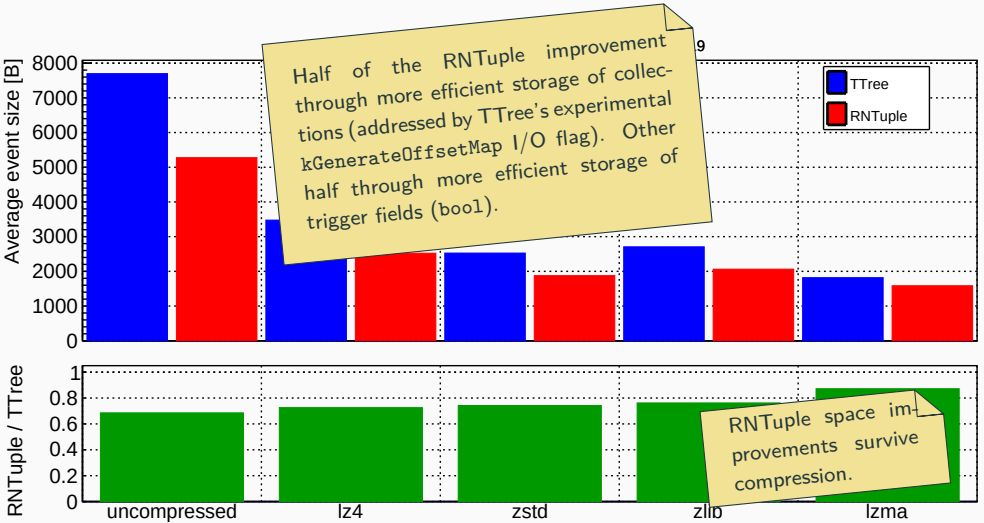
Storage efficiency for events with substructure



Storage efficiency for events with substructure

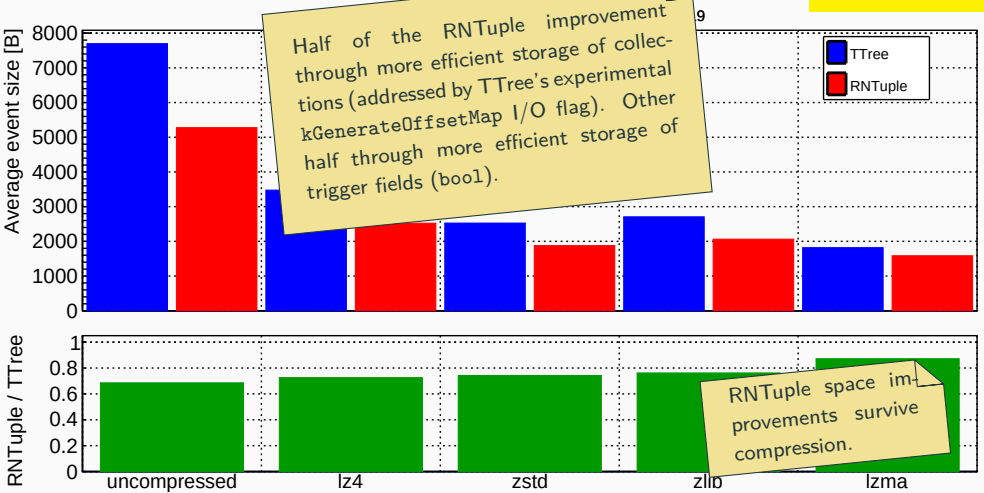


Storage efficiency for events with substructure



Storage efficiency for events with substructure

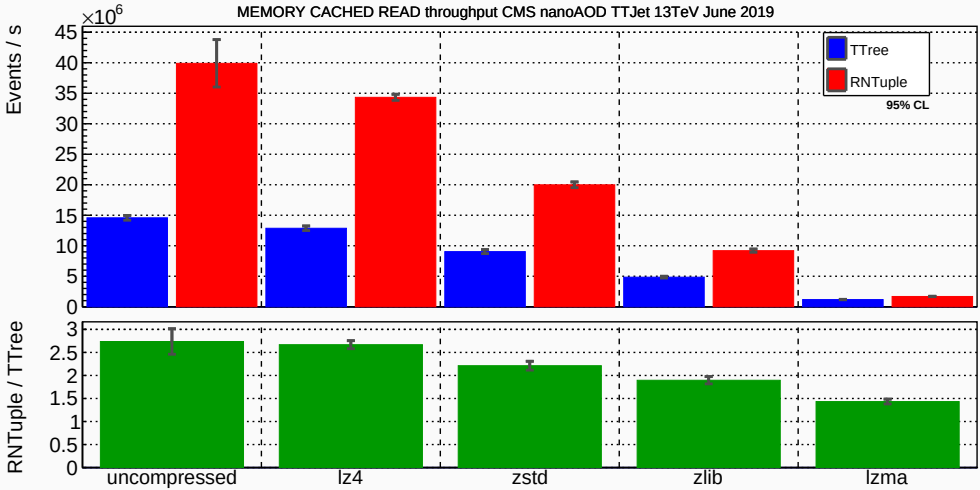
→ see backup slides for "H1" plot



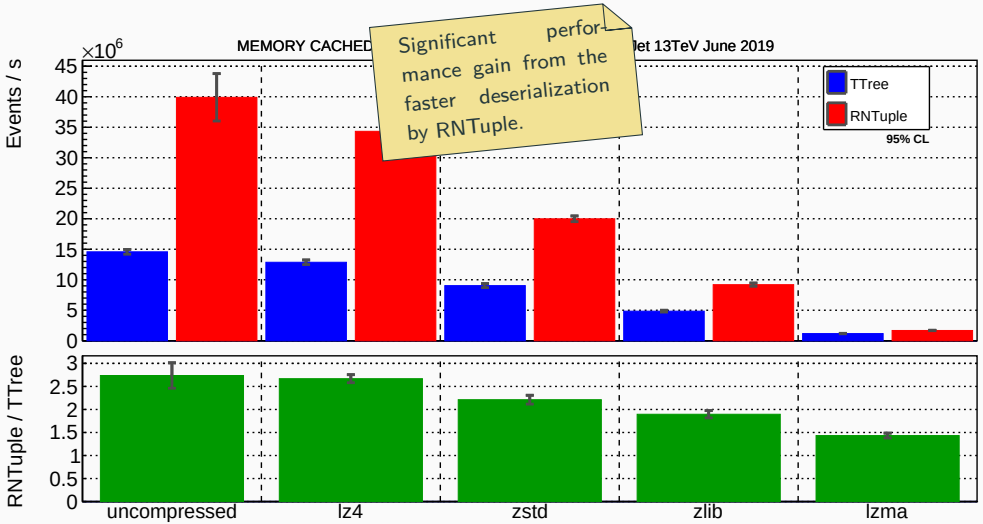
Half of the RNTuple improvement through more efficient storage of collections (addressed by TTree's experimental `kGenerateOffsetMap` I/O flag). Other half through more efficient storage of trigger fields (`bool`).

RNTuple space improvements survive compression.

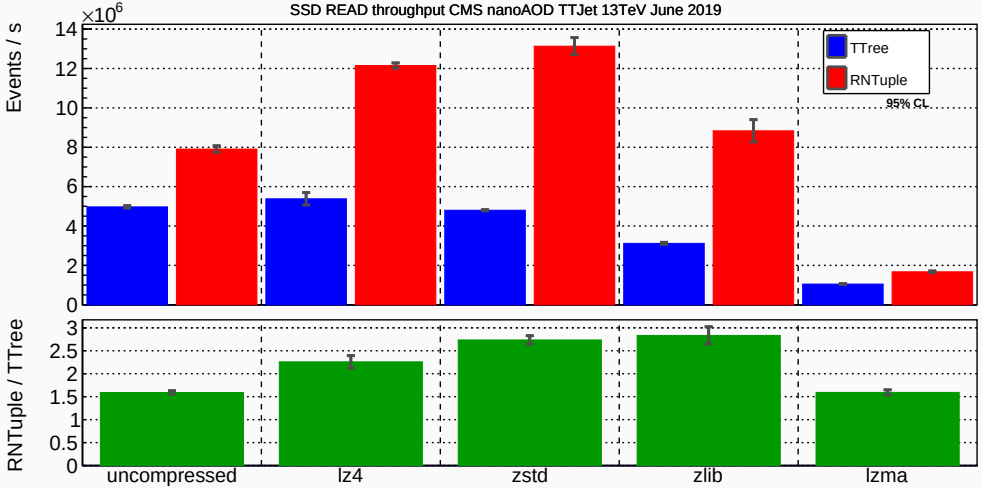
Read speed for warm file system buffers (kernel memory)



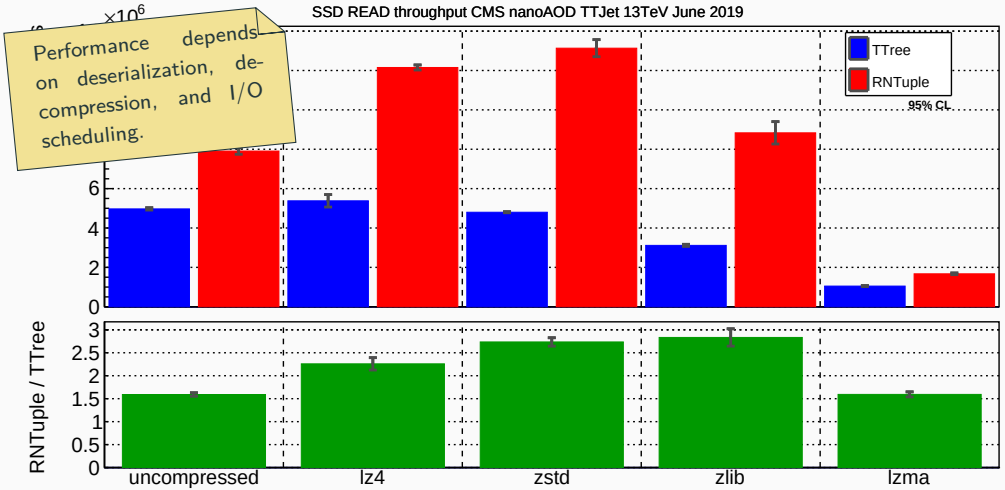
Read speed for warm file system buffers (kernel memory)



Read speed for NVMe SSD

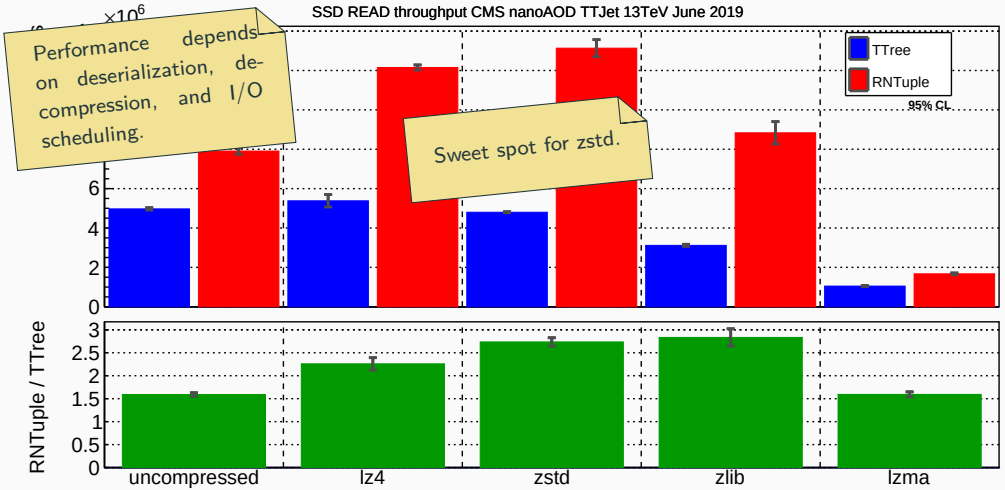


Read speed for NVMe SSD



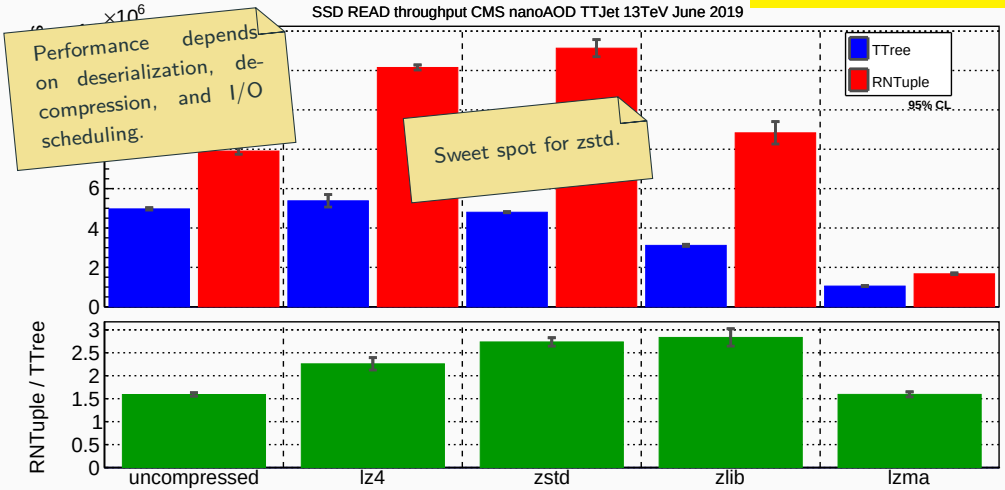
Performance depends on deserialization, decompression, and I/O scheduling.

Read speed for NVMe SSD

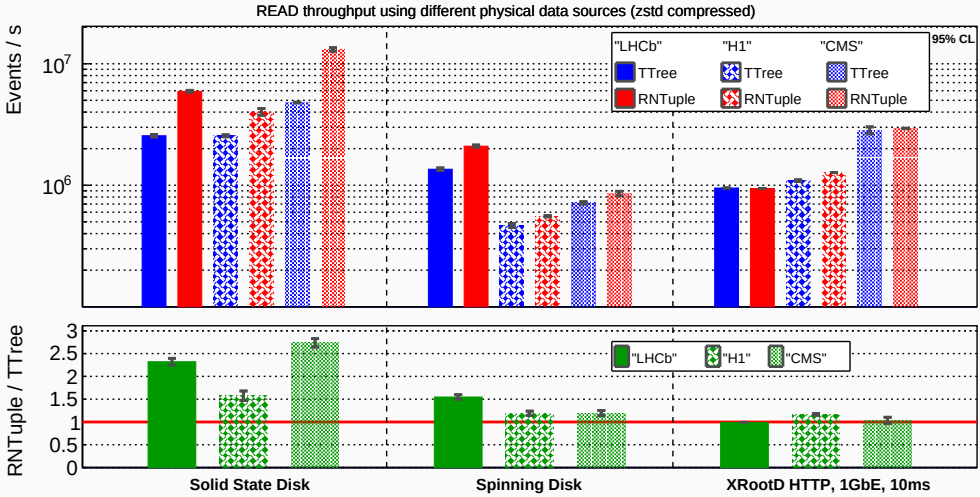


Read speed for NVMe SSD

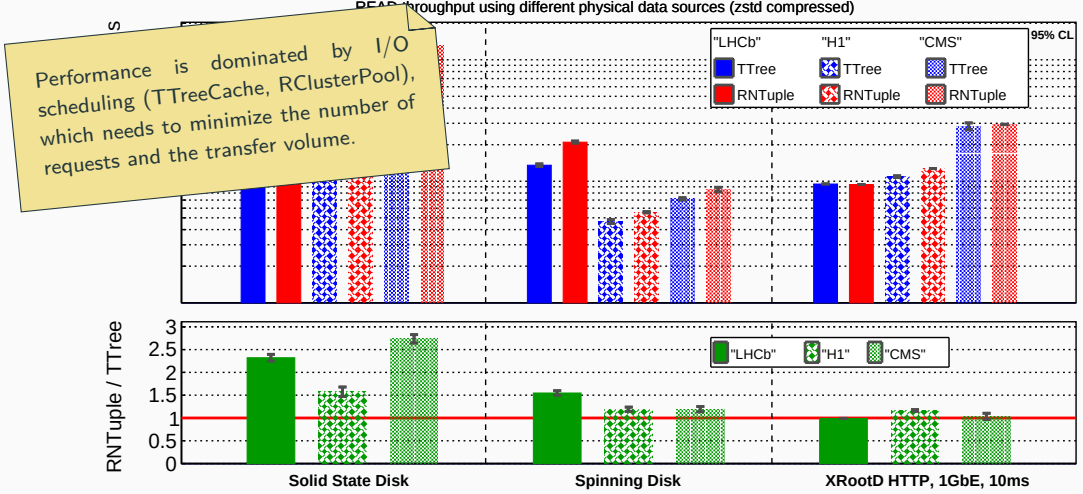
→ Same trend for "LHCb" and "H1" samples



Read speed with different bandwidth and latency profiles

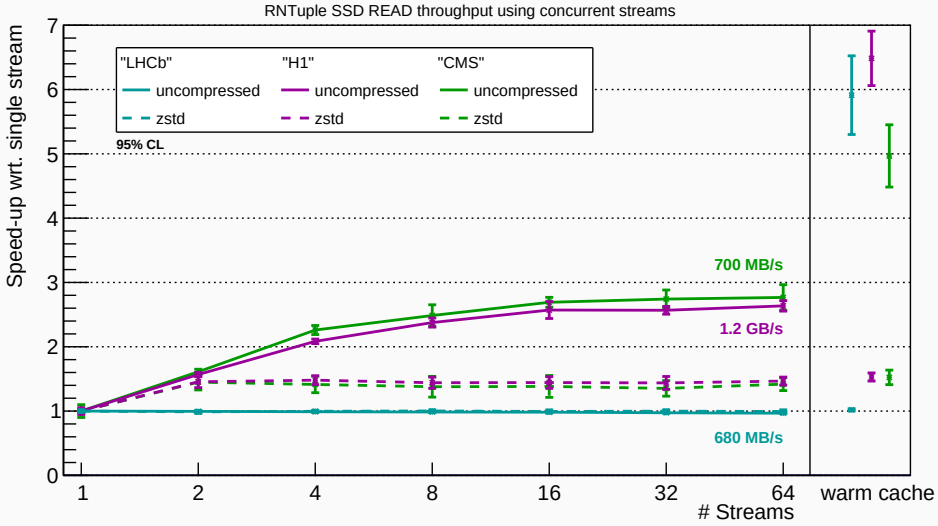


Read speed with different bandwidth and latency profiles

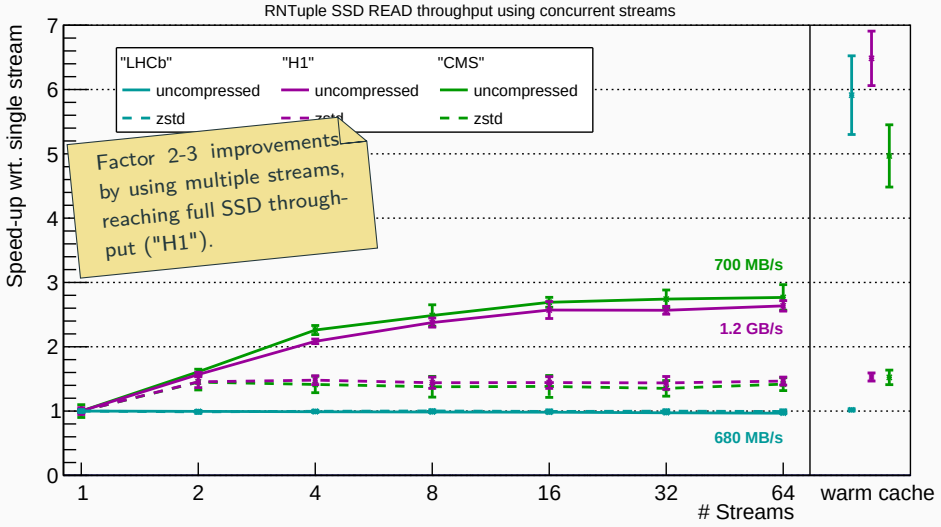


Performance is dominated by I/O scheduling (TTreeCache, RClusterPool), which needs to minimize the number of requests and the transfer volume.

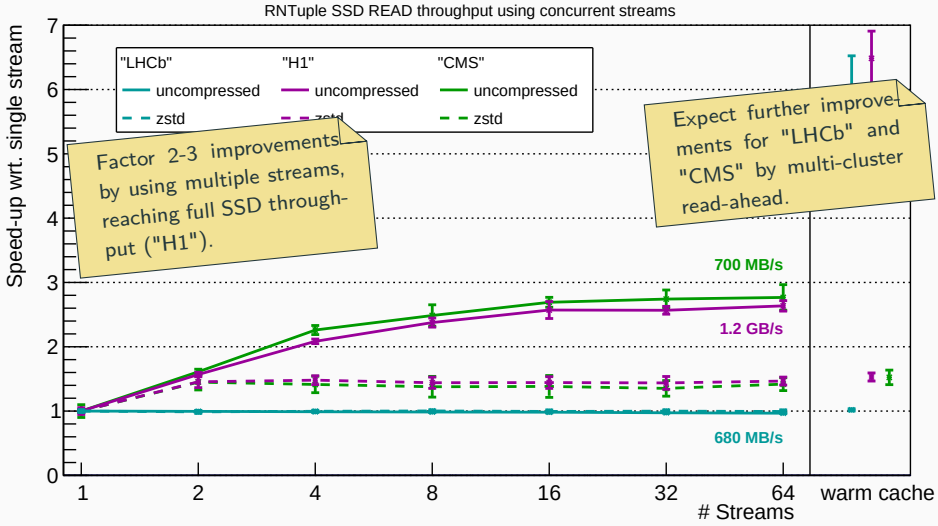
Full exploitation of SSDs by concurrent streams



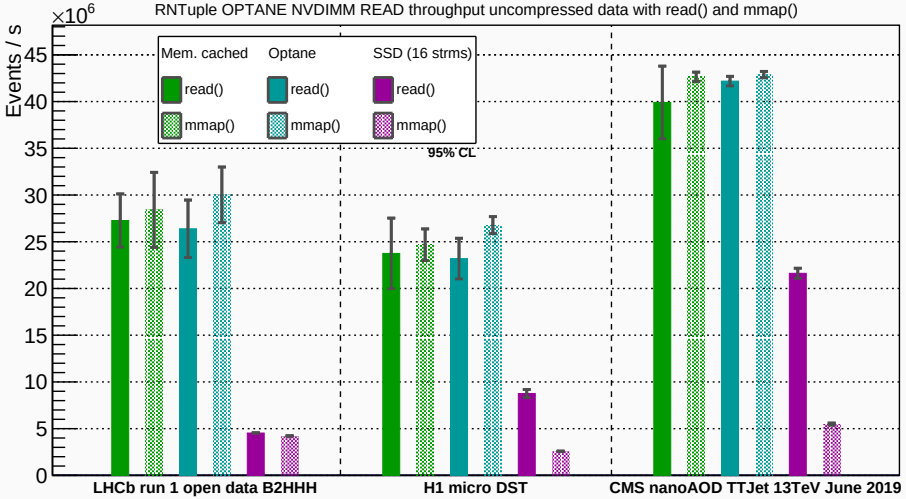
Full exploitation of SSDs by concurrent streams



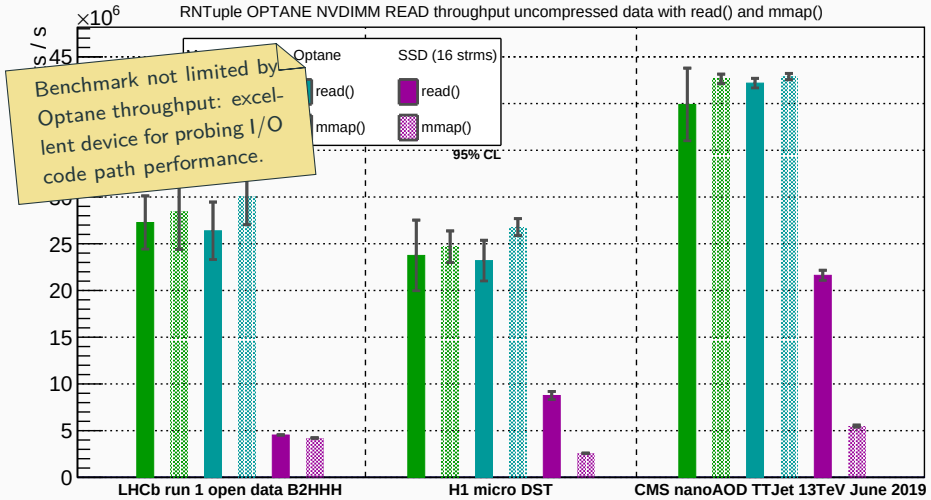
Full exploitation of SSDs by concurrent streams



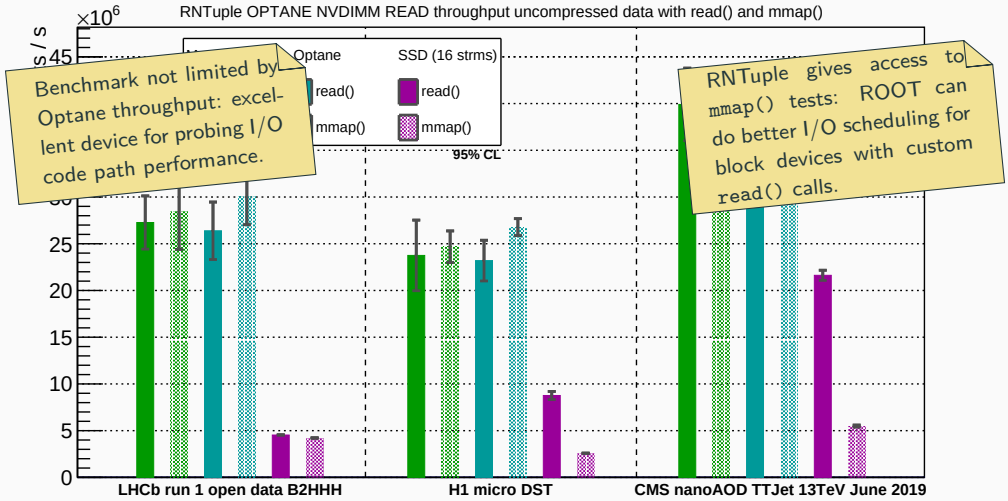
First measurements with Optane NVDIMMs (“App Direct” mode)



First measurements with Optane NVDIMMs (“App Direct” mode)

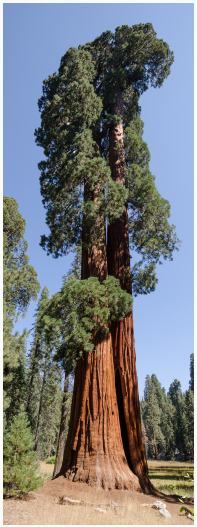


First measurements with Optane NVDIMMs (“App Direct” mode)



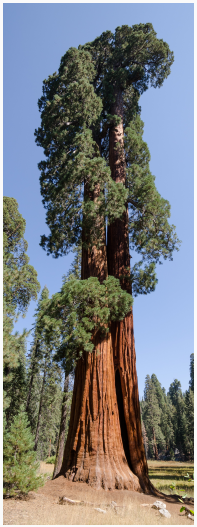
Summary & Outlook

- RNTuple is **exploring the I/O performance frontiers**
- Optimized throughput starting from a blank piece of paper plus 25 years of experience
- Simple, robust, intuitive ROOT7 user interface
- Significant speed-ups for simple event models
- Sneak preview released with ROOT 6.18
  lots of exciting work ahead towards a production-ready I/O subsystem!



Summary & Outlook

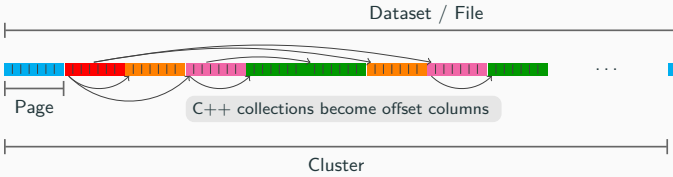
- RNTuple is **exploring the I/O performance frontiers**
- Optimized throughput starting from a blank piece of paper plus 25 years of experience
- Simple, robust, intuitive ROOT7 user interface
- Significant speed-ups for simple event models
- Sneak preview released with ROOT 6.18
  lots of exciting work ahead towards a production-ready I/O subsystem!



Many thanks to CERN openlab and CERN IT for providing test hardware!

Backup Slides

Breakdown of the RNTuple Data Format




```
struct Event {
  int fId;
  vector<Particle> fPtcls;
};
struct Particle {
  float fE;
  vector<int> fIds;
};
```

Cluster

- Block of consecutive complete events
- Unit of thread parallelization (read and write)
- Unit of reading when seeks are expensive
- Typically tens of megabytes

Page

- Unit of memory mapping
- Unit of (de-)compression and (un)packing
- Unit of reading when when seeks are cheap
- Typically tens of kilobytes