RNTuple: Status and Plans of ROOT's I/O Evolution

Jakob Blomer for the ROOT Team

Jefferson Lab Software & Computing Round Table
2021-02-09

ROOT
Data Analysis Framework
https://root.cern

HENP Event Data I/O

Why invest in a **tailor-made I/O system**

TTree & RNTuple

- Capable of storing the **HENP event data model**: nested, inter-dependent collections of data points
- **Performance-tuned** for HENP analysis workflow (columnar binary layout, custom compression etc.)
- **Automatic schema** generation and evolution for C++ (via cling) and Python (via cling + PyROOT)
- Integration with **federated data management** tools (XRootD etc.)
- Long-term **maintenance** and support

Example EDM

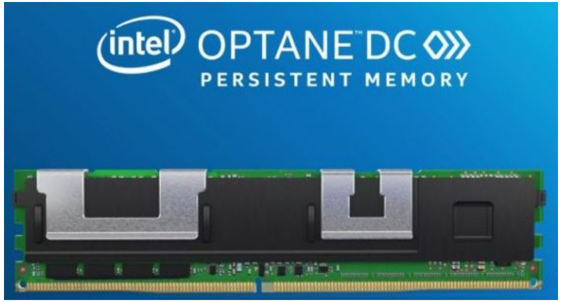
```
struct Event {
    std::vector<Particle> fPtcls;
    std::vector<Track> fTracks;
};

struct Particle {
    float fPt;
    Track &fTrack;
};

struct Track {
    std::vector<Hit> fHits;
};

struct Hit {
    float fX, fY, fZ;
};
```

Motivation for RNTuple

1. HL-LHC challenge: major milestone on the way towards future accelerators and detectors
   - From 300fb^{-1} in run 1-3 to 3000fb^{-1} in run 4-6
   - 10B events/year to 100B events/year
   - Real analysis challenge depends on several factors: number of events, analysis complexity, number of reruns, etc.
     - **As a starting point, preparing for ten times the current demand**

2. Full exploitation of modern storage hardware
   - Ultra fast networks and SSDs: 10GB/s per device reachable (HDD: 250MB/s)
   - Flash storage is inherently parallel ➔ asynchronous, parallel I/O key
   - Heterogeneous computing hardware ➔ GPU should be able to load data directly from SSD, e.g. to feed ML pipeline
   - Distributed storage systems move from POSIX to object stores

At 10GB/s, we have ~3µs to process a 32kB block
➔ CPU optimizations deep into I/O stack



RNTuple Goals

Based on 25+ years of TTree experience, we redesign the I/O subsystem for

- Less disk and CPU usage for same data content
    - 25% smaller files, x2-5 better single-core performance
    - 10GB/s per box and 1GB/s per core sustained end-to-end throughput (compressed data to histograms)
- Native support for object stores (targeting HPC)
- Lossy compression
- Systematic use of exceptions to prevent silent I/O errors

RNTuple Development Plan

~2018-19	~2019-20	~2021-22	~2022-23	~2023-24
Proof of concept	Prototype	First exploitation	Pre-production	Production
• Class design • File format R&D	• Interplay with other ROOT classes • Performance validation	• Interplay with experiment frameworks • Schema evolution	• PB scale test cases • Production tests with last-stage ntuples	• Ready to use for new runs

We see RNTuple as a Run 4 technology

Available now in `ROOT::Experimental`

Note: TTree technology will remain available for the 1EB+ existing data sets

RNTuple Class Design

Seamless transition from TTree to RNTuple

Event iteration
Reading and writing in event loops and through RDataFrame
RNTupleDataSource, 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

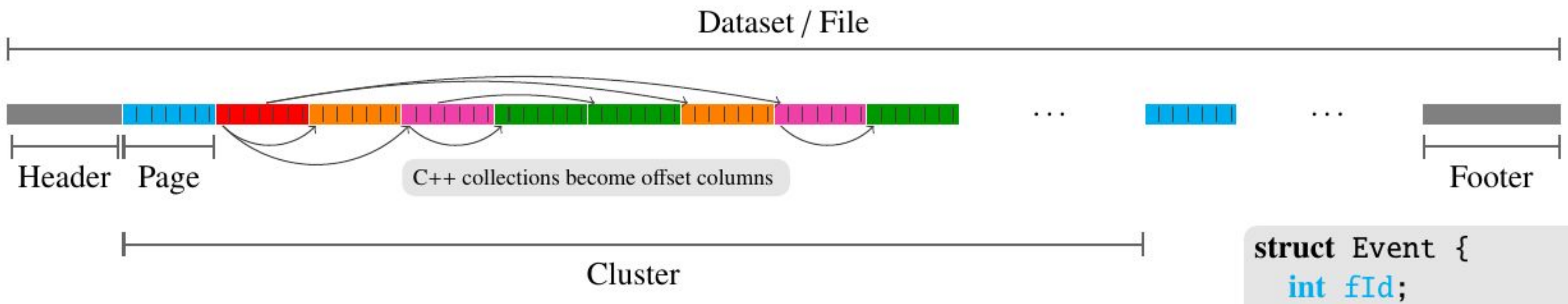
Storage layer / byte ranges
RPageStorage, RCluster, RNTupleDescriptor

Modular storage layer that supports files as data containers but also file-less systems (object stores)

Approximate translation between TTree and RNTuple classes:

TTree	≈	RNTupleReader RNTupleWriter
TTreeReader	≈	RNTupleView
TBranch	≈	RField
TBasket	≈	RPage
TTreeCache	≈	RClusterPool

RNTuple Format Breakdown



Approximate translation between TTree and RNTuple concepts:

Basket	≈	Page
Leaf	≈	Column
Cluster	≈	Cluster

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

Cluster:

- Block of consecutive complete events
- Independent from each other, e.g. can be distributed across machines
- O(100MB)

Page:

- Unit of memory mapping or (de)compression
- Parallel (de)compression `ROOT::EnableImplicitMT()`
- O(100kB)

RNTuple Format Evolution

- Key binary layout changes wrt. TTree
  - More efficient nested collections
  - More efficient boolean values (bitfield), interesting for trigger bits
  - experimenting with "split floats"
  - Little-endian values (allows for mmap())

Implementation uses templates to slash memory copies and virtual function calls in common I/O paths

- Supported type system
  - Boolean
  - Integers, floating point
  - std::string
  - std::vector, std::array
  - std::variant
  - User-defined classes
  - More classes planned (e.g. std::chrono)

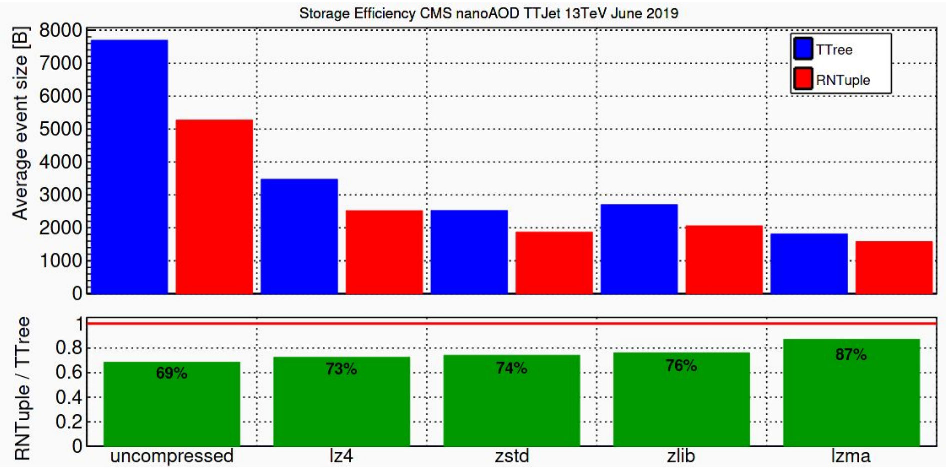
Fully composable (including aggregation, inheritance) within the supported type system

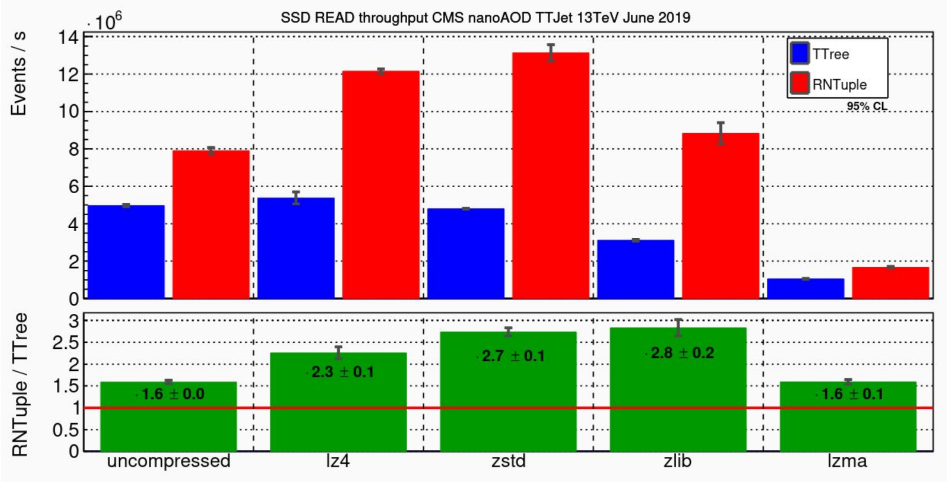
Performance Evaluation Samples

LHCb run 1 OpenData	H1 DST	CMS 2019 nanoAOD	ATLAS 2020 open data
B mass spectrum	ROOT standard benchmark	Dimuon spectrum	H → gg
Fully flat EDM	EDM with collections	EDM with collections	Set of std::vector
Dense reading (>75%, 18/26 features)	Medium dense reading (~10%, 16/152 features)	Sparse reading (<1%, 6/1479 features)	Medium dense reading (~25%, 12/81 features)
8.5 million events 24k selected events	2.8 million events 75k selected events	1.6 million events 141k selected events	7.8 million events 76k selected events
1.5GB	3.4GB	8GB	3GB

We'd be happy to add a toy analysis on a simple EDM typical for Nuclear Physics

Selected RNTuple Benchmarks (2019)



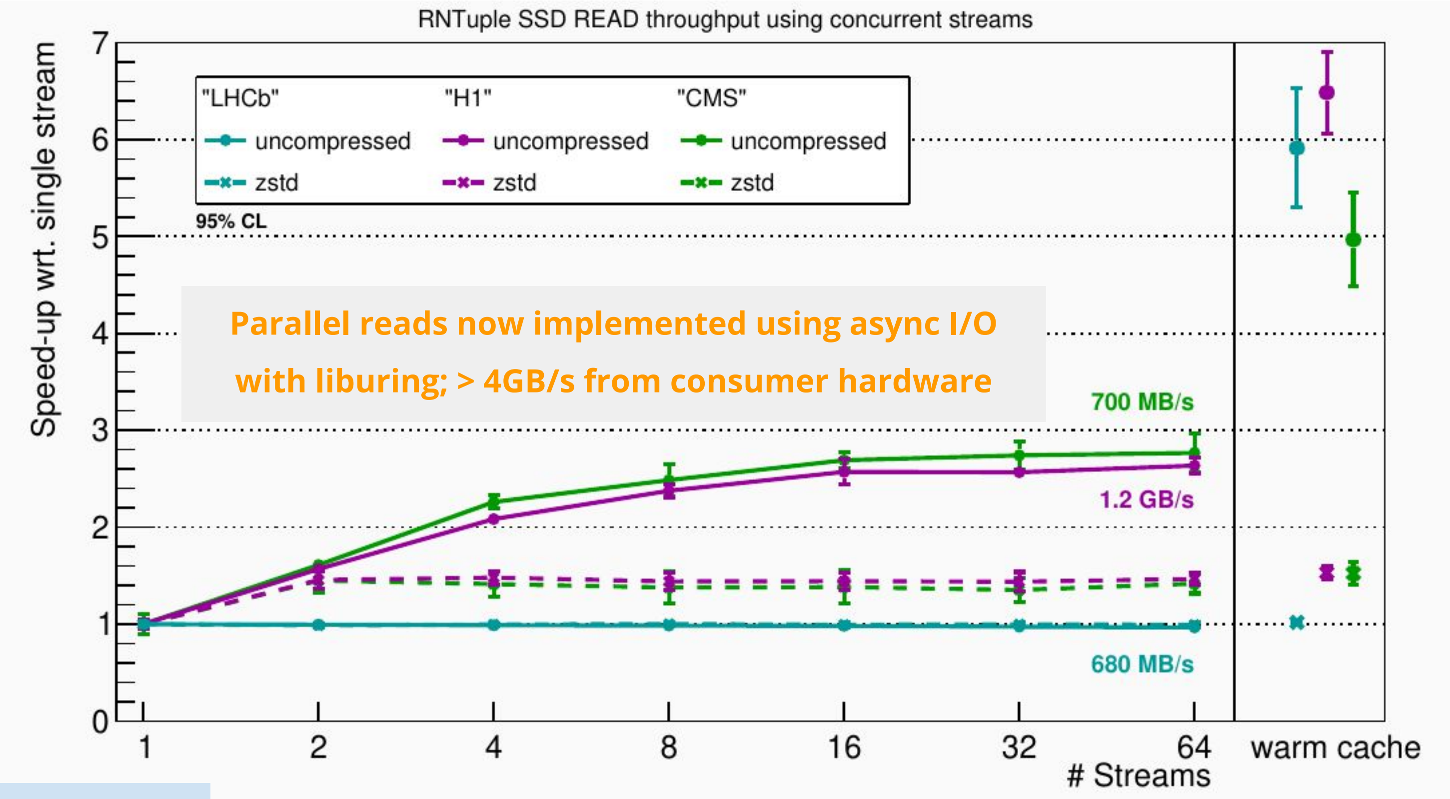


Comparing end-to-end performance (data → histograms)

- Substantially smaller files and better performance, already on single-core
- Continuous effort to improve performance
- Updated comparison with HDF5 planned for 2021 (→ 2017 results)

→ https://github.com/jblomer/iotools

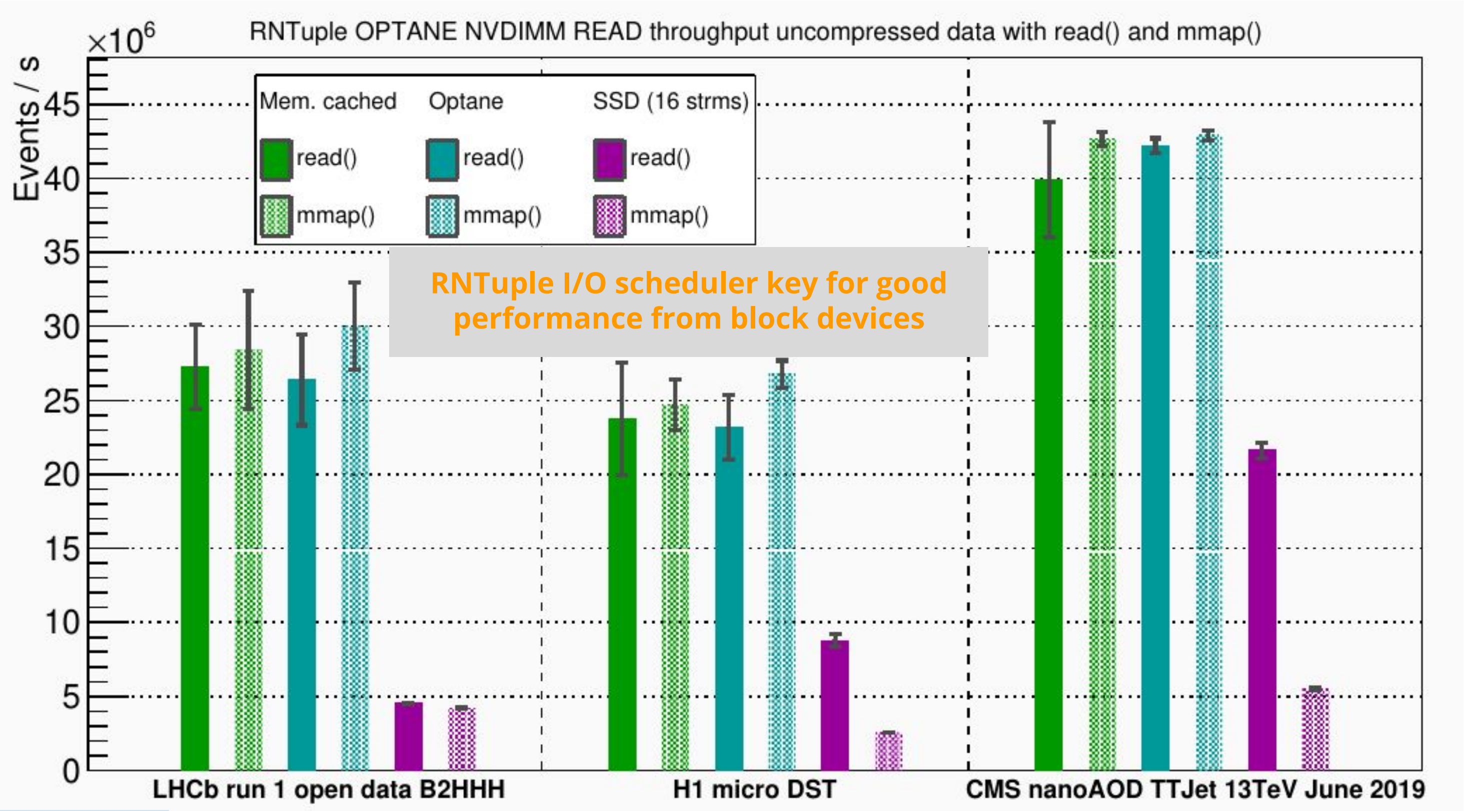
Selected RNTuple Benchmarks (2019)



Parallel reads now implemented using async I/O with liburing; > 4GB/s from consumer hardware

→ https://github.com/jblomer/iotools

Selected RNTuple Benchmarks (2019)



RNTuple I/O scheduler key for good performance from block devices

→ https://github.com/jblomer/iotools

R&D: RNTuple and object stores

Object store technology

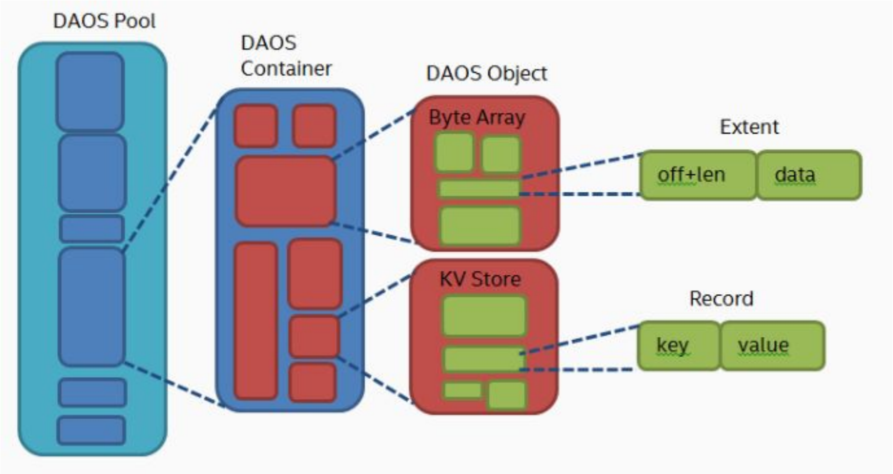
- Very scalable, distributed storage
  - Popular because it overcomes POSIX I/O limitations of shared cluster file systems
  - Standard file system access only provided through slow compatibility layer
- Relevant for exascale HPC systems: Argonne's Aurora going to provide 220 PB of Intel DAOS object storage

In RNTuple

- Native support for object store
- Goal: avoid transient copies to other file formats in HPCs

Mapping RNTuple → DAOS

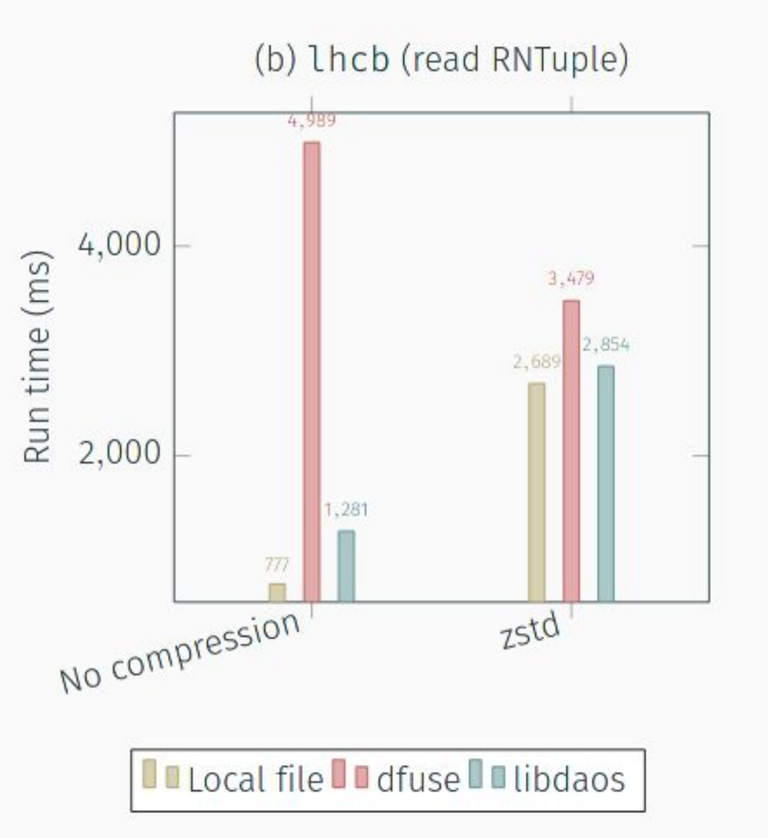
```
auto ntuple = RNTupleReader::Open("DecayTree", "daos://41adf800b537...");
```



DAOS object: a key–value store with locality. The key is split into **dkey** (distribution key) and **akey** (attribute key).

RNTuple → Container
Cluster → Object
Page group → dkey
Page → akey

DAOS Preliminary Read Results



Preliminary results

- 1 client, 3 servers (provided by CERN openlab)
- Significantly faster read with RNTuple than with dfuse compatibility layer
- ~1GB/s throughput (should be even higher, still under investigation)

Ongoing: NanoAOD RNTuples

IRIS-HEP is funding a project to generate NanoAODs in RNTuple format!

- Validates that RNTuple can handle complete nanoAOD EDM
- Provides first experience with framework integration
- Allows for tuning multi-threaded write
- Allows for large-scale comparison between TTree and RNTuple

Goal

A CMSSW output module to write NanoAODs in RNTuple format.

Try out RNTuple

- RNTuple is available in ROOT::Experimental

- Build with `-Droot7=on -DCMAKE_CXX_STANDARD=14`
- Check out the RNTuple Tutorials...
- ... as well as toy analyses and benchmarks

- Questions? Contact us at the ROOT forum

Summary

RNTuple: ROOT's I/O R&D aiming at a **leap in data throughput**

- Updated (backwards incompatible) data format for next-generation event I/O
- Expect ~25% smaller files,
  x2-5 better single-core throughput on SSD
- Aims at using modern I/O devices to the full capacity
- Modern, robust API (e.g., thread-friendly, systematic use of exceptions)
- Entering first exploitation phase (➡ 2021 PoW)