RNTuple IO Design and Object Stores

Vincenzo Eduardo Padulano for the ROOT team

Compute & Accelerator Forum - High Performance IO for HEP
10.04.2024

Topics

- RNTuple design
- Support for Object Stores

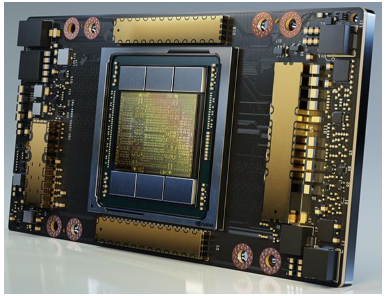
Design

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

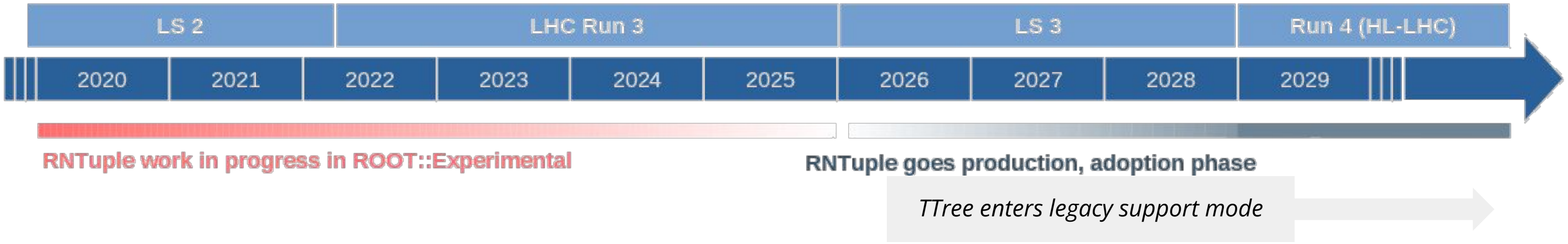
At 10GB/s, we have ~3µs to process a 32kB block
➔ CPU optimizations deep into I/O stack



RNTuple introduction

Redesigned I/O subsystem, based on **25+ years** of **TTree** experience

- **Less** disk and CPU **usage**
- **Efficient** support of **modern hardware**
- Transparent **file-less** storage
- **Covering** all of today's **TTree** use cases
- **Binary format** defined in a dedicated specification
- More info from the recent RNTuple workshop



RNTuple Workshop

- 6-7 November 2023 @ CERN (indico)
- ~30 participants
  - ROOT team, LHC core computing devs, IT, HEP-CCE
- One presentation by each experiment
  - ATLAS most advanced, then CMS, LHCb and ALICE
- Misc: DUNE, ATLAS metadata, SoA in CMS

Schedule Presented to LHCC, Updated

~2018-19 — Proof of concept
- ✅ Architecture
- ✅ Review on state-of-the-art
- ✅ First prototypes

~2019-20 — Prototype
- ✅ Adoption in ROOT::Experimental
- ✅ I/O scheduler for local and remote access
- ✅ Performance validation

~2021-22 — First exploitation
- ⛅ Object store support
  - ✅ DAOS (HPC)
  - ⛅ S3 (Cloud)
- ✅ RNTuple version 1 spec
- ⛅ RNTupleLite
- ⛅ Schema evolution
- ✅ Disk-to-disk conversion
- 🕑 Virtual data sets for skims and selections
- ✅ First exposure to frameworks:
  - ✅ CMSSW nanoAOD output module
  - ✅ Prototyping by ATLAS, CMS, LHCb I/O experts

~2022-23 — Pre-production
- ✅ RDataFrame bulk processing
- ✅ Debugging and inspection tools
- 🕑 Metadata API
- ✅ Special use case support: e.g. backfill, in-memory adapters
- ✅ XRootD support
- ⛅ Validation of feature coverage
- ✅ Training experiments' core developers
- ⛅ Large-scale experiment benchmarks

~2023-24 — Production
- ⛅ PB scale tests
- 🕑 Automatic optimization features
- ✅ Low-precision floats
- 🕑 ML Training: direct GPU transfer
- 🕑 End-user training
- ⛅ Training and support for code and data migration

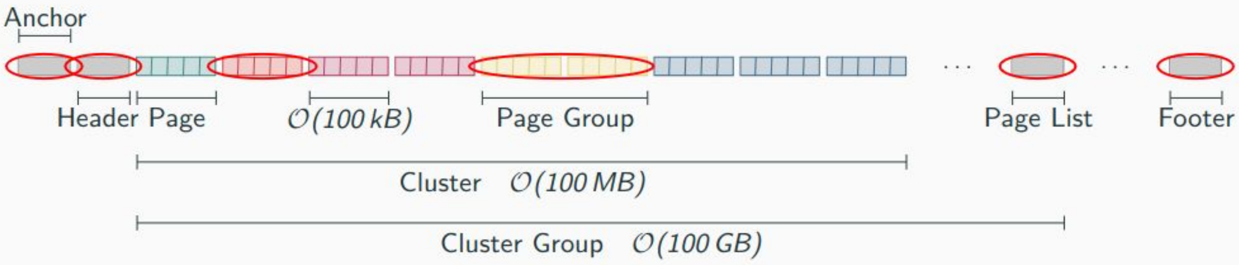
✅ = available
⛅ = under development
🕑 = programme of work
— = in collaboration with users/experiments

Work items defined: Nov 2021
Development state: Apr 2024

Growing importance of coordination & collaboration with experiment I/O experts

RNTuple data layout

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



Read pattern:

1. File open: read anchor, header, footer (once)
2. Read page list (once per cluster group)
3. Background thread: read-ahead page groups for the next *k* clusters
   in vector reads, close-by byte ranges get coalesced

RNTuple Binary Format Walk-Through

Benefits of new binary format

- More efficient storage of collections and boolean values
- Addition of new basic types, e.g. f16
- Little-endian numbers: memory mappable on most contemporary platforms
- Type-based encoding: e.g. zig-zag for signed ints, bit packing for bools, etc.
- Split storage for arbitrarily nested collections
- More scalable meta-data, better memory control
- New default compression: zstd
- Format independent of TFile

```
root [1] .ls
TFile**          basic2.root
 TFile*          basic2.root
  KEY: TTree     ntuple;1        data from ascii file
  KEY: ROOT::Experimental::RNTuple      imported;1      object title
root [2] _file0->Map()
20231028/012556  At:100     N=118       TFile
20231028/012556  At:218     N=3824      TBasket        CX =  1.06
20231028/012556  At:4042    N=3826      TBasket        CX =  1.06
20231028/012556  At:7868    N=3754      TBasket        CX =  1.08
20231028/012556  At:11622   N=511       TTree          CX =  3.55
20231028/013026  At:12133   N=65        FreeSegments
Address = 12198 Nbytes = -4750   =====G A P===========
20231028/013026  At:16948   N=176       RBlob          CX =  1.66
20231028/013026  At:17124   N=3745      RBlob          CX =  1.08
20231028/013026  At:20869   N=3728      RBlob          CX =  1.08
20231028/013026  At:24597   N=3517      RBlob          CX =  1.15
20231028/013026  At:28114   N=126       RBlob          CX =  1.32
20231028/013026  At:28240   N=128       RBlob          CX =  1.30
20231028/013026  At:28368   N=134       ROOT::Experimental::RNTuple
20231028/013026  At:28502   N=185       KeysList
20231028/013026  At:28687   N=4909      StreamerInfo   CX =  3.11
20231028/013026  At:33596   N=1         END
root [3]
```

RNTuple Limits

Limit	Value	Reason / Comment
Volume	1-10 PB (theoretically more)	Assuming 10k cluster groups of 10k clusters of 10-100MB each
Number of elements, entries	2^64	Using default (Split)Index64, otherwise 2^32
Cluster & entry size	8TB (depends on pagination)	Assuming limit of 4B pages of 4kB each
Page size	2B elements, 256MB-2GB	#elements * element size, 2GB limit from locator
Element size	8kB	16bit for number of bits per element
Number of column types	64k	16bit for column type
Envelope size	2^48B (~280TB)	Envelope header encoding
Field / type version	4B	Field meta-data encoding
Number of fields, columns	4B (foreseen: <10M)	32bit column / field IDs, list frame limit
Number of clusters per group	4B (foreseen: <10k)	List frame limits, cluster group summary encoding
Number of pages per cluster per column	4B	List frame limits

Note: RNTuple in addition is subject to limits from TFile / object store backend

RNTuple Utilities

Convert your existing TTree to RNTuple:

```
#include <ROOT/RNTupleImporter.hxx>
using ROOT::Experimental::RNTupleImporter;

auto importer = RNTupleImporter::Create(
   "Events",
   "myNanoAOD.ttree.root",
   "myNanoAOD.rntuple.root");

// Optional
importer->SetNTupleName("EventsNTuple");

auto writeOptions = importer->GetWriteOptions();
// Optional, default is zstd level 5
auto algo = RCompressionSetting::EAlgorithm::kLZMA;
writeOptions.SetCompression(algo, 7);
importer->SetWriteOptions(writeOptions);

importer->Import();
```

RNTupleImporter docs and tutorial

Get detailed storage information for your RNTuple:

```
#include <ROOT/RNTupleInspector.hxx>
using ROOT::Experimental::RNTupleInspector;

auto inspector = RNTupleInspector::Create(
   "EventsNTuple", "myNanoAOD.rntuple.root");

std::cout << "My NanoAOD is compressed using "
          << inspector->GetCompressionSettingsAsString()
          << std::endl;
inspector->PrintColumnTypeInfo();
```

```
My NanoAOD is compressed using lzma (level 7)
 column type | count | # elems    | compr. bytes | uncompr. bytes
-------------|-- ----|------------|--------------|-----------------
SplitIndex64 | 5     | 267230990  | 84109056     | 2137847920
 SplitReal32 | 45    | 3856668029 | 11402474398  | 15426672116
  SplitInt32 | 15    | 1436663181 | 147427186    | 5746652724
```

RNTupleInspector docs

RNTupleMetrics

```
auto tree = file->Get<TTree>("tree");
TTreePerfStats *ps = new TTreePerfStats("ioperf", tree);
// …
ps->Print();
```

```
TreeCache = 30 MBytes
N leaves  = 26
ReadTotal = 749.412 MBytes
ReadUnZip = 1137.82 MBytes
ReadCalls = 524
ReadSize  = 1430.176 KBytes/read
Readahead = 256 KBytes
Readextra =  0.00 per cent
Real Time =   2.090 seconds
CPU  Time =   1.550 seconds
Disk Time =   0.724 seconds
Disk IO   = 1034.508 MBytes/s
ReadUZRT  = 544.310 MBytes/s
ReadUZCP  = 734.076 MBytes/s
ReadRT    = 358.504 MBytes/s
ReadCP    = 483.492 MBytes/s
```

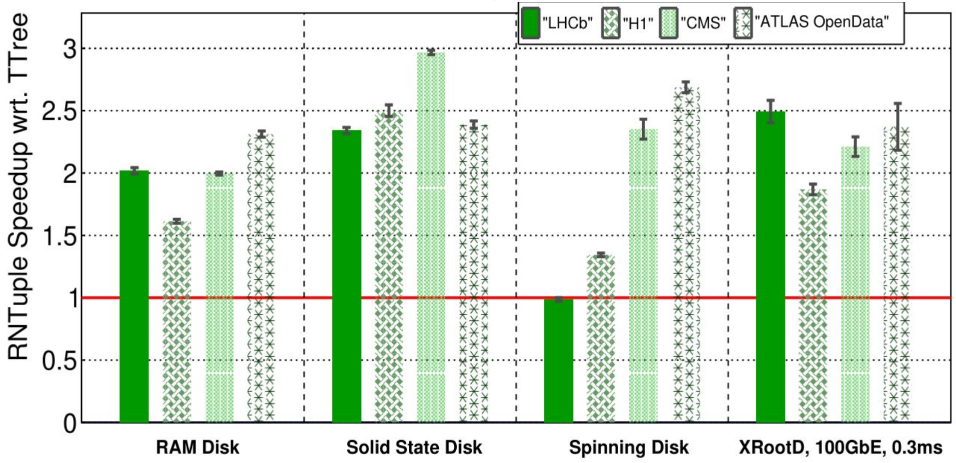
```
auto anchor = file->Get<RNTuple>("ntpl");
auto reader = RNTupleReader::Open(anchor);
reader->EnableMetrics();
// …
reader->PrintInfo(ENTupleInfo::kMetrics);
```

```
RNTupleReader.RPageSourceFile.nReadV||number of vector read requests|21
RNTupleReader.RPageSourceFile.nRead||number of byte ranges read|834
RNTupleReader.RPageSourceFile.szReadPayload|B|volume read from storage (required)|731470154
RNTupleReader.RPageSourceFile.szReadOverhead|B|volume read from storage (overhead)|180996722
RNTupleReader.RPageSourceFile.szUnzip|B|volume after unzipping|1129407576
RNTupleReader.RPageSourceFile.nClusterLoaded||number of partial clusters preloaded from storage|21
RNTupleReader.RPageSourceFile.nPageLoaded||number of pages loaded from storage|17175
RNTupleReader.RPageSourceFile.nPagePopulated||number of populated pages|17175
RNTupleReader.RPageSourceFile.timeWallRead|ns|wall clock time spent reading|337259128
RNTupleReader.RPageSourceFile.timeWallUnzip|ns|wall clock time spent decompressing|527901157
RNTupleReader.RPageSourceFile.timeCpuRead|ns|CPU time spent reading|1355967000
RNTupleReader.RPageSourceFile.timeCpuUnzip|ns|CPU time spent decompressing|1373490000
RNTupleReader.RPageSourceFile.bwRead|MB/s|bandwidth compressed bytes read per second|2705.536486
RNTupleReader.RPageSourceFile.bwReadUnzip|MB/s|bandwidth uncompressed bytes read per second|3348.782827
RNTupleReader.RPageSourceFile.bwUnzip|MB/s|decompression bandwidth of uncompressed bytes per second|2139.430007
RNTupleReader.RPageSourceFile.rtReadEfficiency||ratio of payload over all bytes read|0.801640
RNTupleReader.RPageSourceFile.rtCompression||ratio of compressed bytes / uncompressed bytes|0.647658
```

Performance improvements **across** the board

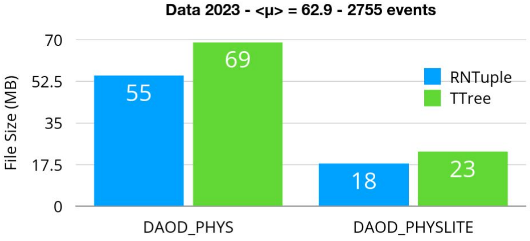
Storage **saving** and runtime **decrease**

Time to plot: RDataFrame analysis with RNTuple input data



RNTuple: A Quick Look at DAOD Performance

- **Current studies indicate about 20+% storage savings is possible in DAODs**
  - It's important to note TTree is heavily optimized over the last 20 years
  - Similar optimization studies will be carried out for RNTuple prior to production



Sample DAOD_PHYSLITE in RNTuple

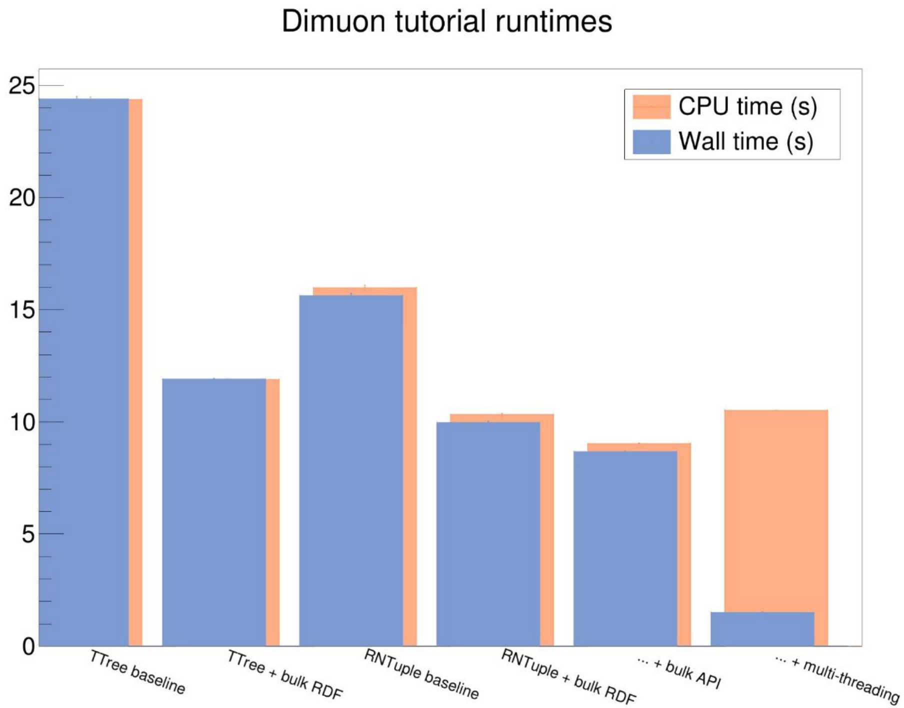
```
NTUPLE:       RNT:CollectionTree
Compression: 505
------------------------------------------
  # Entries:          2755
  # Fields:           1348
  # Columns:          1035
  # Alias Columns:    0
  # Pages:            3444
  # Clusters:         2
  Size on storage:    18593394 B
  Compression rate:   5.48
  Header size:        7213 B
  Footer size:        23202 B
  Meta-data / data:   0.002
```

ACAT `24, S. Mete

CHEP '23, J. Blomer

RDataFrame + RNTuple

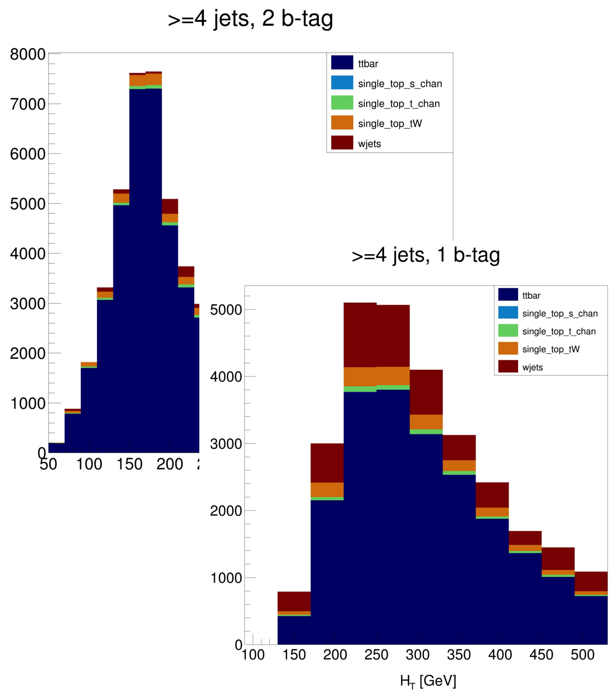
- Bulk, asynchronous I/O **and** bulk processing
    - Hide network latency
    - Enable SIMD on CPU, GPU offloading



RDataFrame + Analysis Grand Challenge

AGC – HEP analysis benchmarks

- In various implementations, including with RDataFrame
- In particular: $t\bar{t}$ analysis based on CMS Open Data



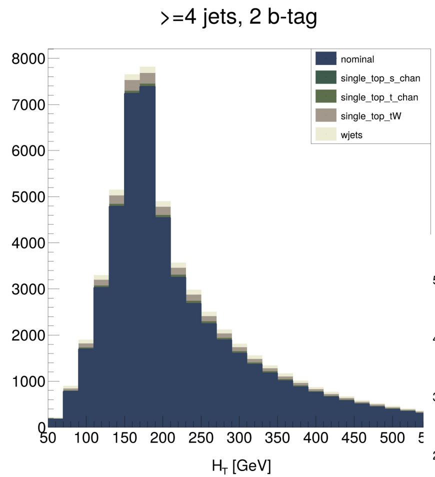
Achievements:

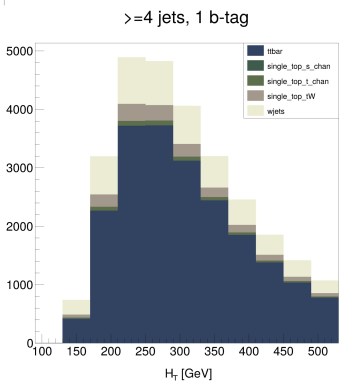
- Tagged RDF AGC v1
- Implemented v2 ML inference (via FastForest)
- Local, multi-thread and distributed Dask execution
- Bin-by-bin agreement of output histograms
- Works with TTree inputs and also with RNTuple

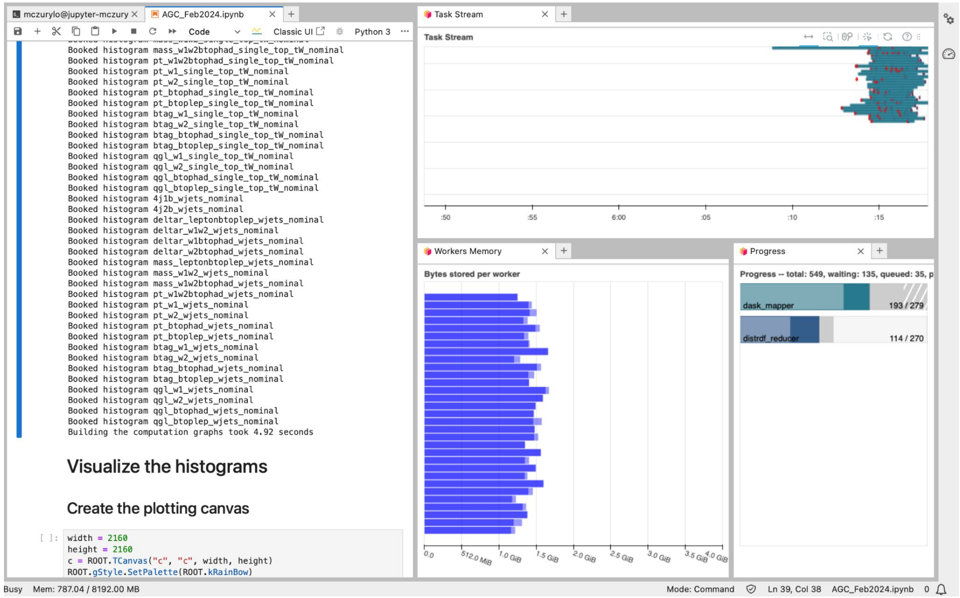
Distributed AGC with TTree and RNTuple – user side

The only change for the user - the ROOT input file!

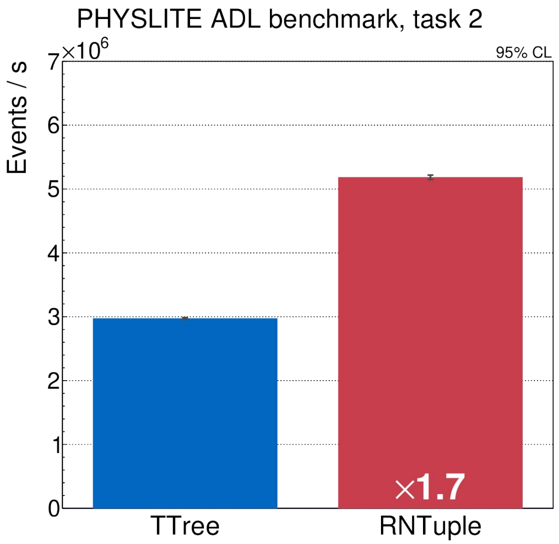
```
REMOTE_DATA_PREFIX: str = "root://eospublic.cern.ch//eos/root-eos/AGC/"
REMOTE_DATA_PREFIX: str = "root://eospublic.cern.ch//eos/root-eos/AGC/rntuple/"
```



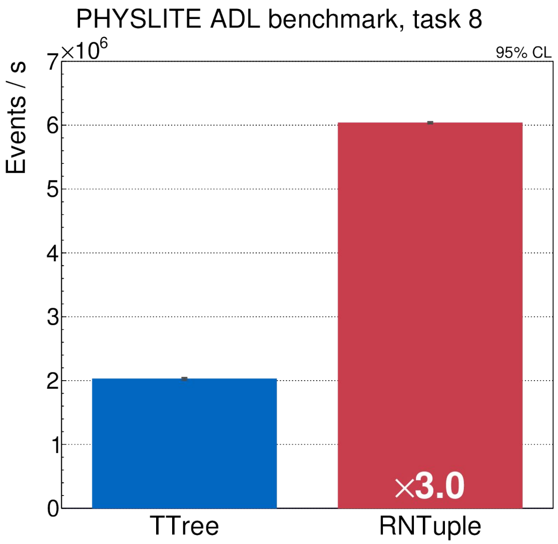




ADL Benchmarks



1 branch/top-level field read,
no cuts/calculations applied



12 branches/top-level fields read,
some (basic) cuts/calculations applied

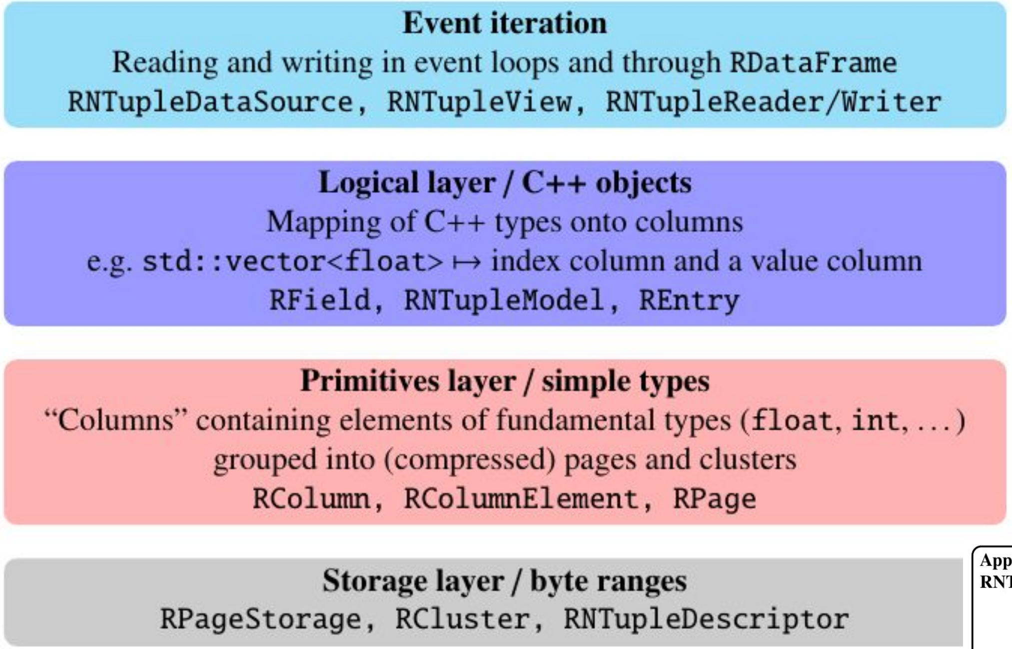
- **A**nalysis **D**escription **L**anguage Benchmarks
- Originally implemented for CMS NanoAOD, adapted to PHYSLITE
- **zstd** compressed
- Single-core throughput with **RDataFrame**

PHYSLITE ADL repository

RNTuple API Review

- A new effort in collaboration with HEP-CCE
- Expose RNTuple API for external experts' validation
  - Should help us assess functionality, consistency, safety, and usability in the context of HEP experiment software frameworks
  - The results of this review will guide further developments

RNTuple Class Design



- General design guidelines
  - Following C++ core guidelines
  - Use of exceptions (`RException`)
  - Conditionally thread-safe
  - Compile-time type-safe interfaces, runtime type-safe interfaces and `void *` interfaces
  - Shared pointers for values to be (de-)serialized
    - With option to pass raw pointers
  - Separation of read and write path

- For reading from files, RNTuple uses `RRawFile`, i.e. no dependency on TFile or TBuffer. RRawFile has plugins for HTTP and XRootD

Approximate translation between TTree and RNTuple classes:

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

API Walk-Through

- `RNTuple`
  - Anchor, references RNTuple data
  - Can be used as in input to other classes, e.g. RNTupleReader
- `RPageSource / RPageSink`
  - Reads and writes pages from the storage backend (file, object store, etc)
  - No concept of entries, only columns
  - Not user-facing
- `RNTupleDescriptor`
  - Gives access to the on-disk meta-data

```
auto anchor = file->Get<RNTuple>("ntpl");
auto reader = RNTupleReader::Open(anchor); // unique_ptr
const auto &entry = reader->GetModel().GetDefaultEntry();
auto pt = entry.GetPtr<std::vector<double>>("pt");
reader->LoadEntry(0);
// See writer example for the void * API using entries
```

```
const auto &descriptor = reader->GetDescriptor();
for (const auto &fieldDesc : desc->GetTopLevelFields()) {
  std::cout << fieldDesc.GetFieldName() << ": "
            << fieldDesc.GetTypeName() << std::endl;
}
```

API Walk-Through

- `RField<T>`
  - Central class: connects the in-memory representation of data to its on-disk representation
  - Can connect to a page source or sink
- `RField::RValue`
  - Connects a value in memory to a corresponding field
  - Used to safely read/write data (prevents mistakenly reading/writing from wrong field)
- `RNTupleModel`
  - Schema representation as a tree of fields
  - Can create entries
- `REntry`
  - Represents a row: values for the top-level fields of a model
- `RNTupleReader, RNTupleWriter`
  - Event iteration for reading/writing

```
auto fieldEta =
  std::make_unique<RField<std::vector<double>>>("eta");
auto fieldPt =
  RFieldBase::Create("pt", "std::vector<double>").Unwrap();

auto  value = fieldPtr->CreateValue();
auto ptSharedPtr = value.GetPtr<std::vector<double>>();
auto *pt = fieldPt->CreateObject<std::vector<double>>().release();
```

```
auto model = RNTupleModel::Create();
model->AddField(std::move(fieldEta));
model->AddField(std::move(fieldPt));
{
  auto writer = RNTupleWriter::Append(std::move(model), "ntpl", *f);
  auto entry = writer->CreateEntry()
  entry->BindRawPtr("eta", myEta);
  entry->BindRawPtr("pt", myPt);
  writer->Fill(*entry);
}
```

Support for Object Stores

Why object stores?

In a highly-parallel setting, object stores align well with our requirements:

- Extremely scalable
- Widely deployed in cloud service providers

Where?

- HPC: Intel DAOS
- Cloud: Amazon S3, Microsoft Azure Blob, Google Cloud

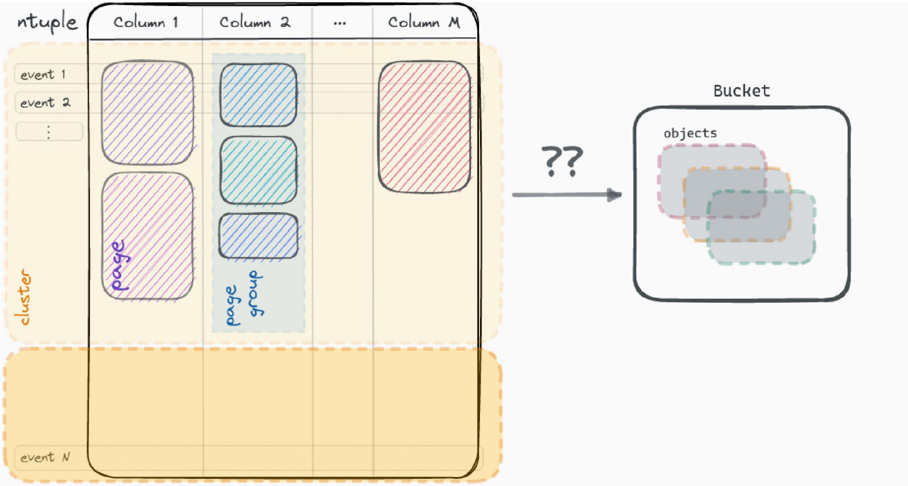
Why?

- Better scalability in parallel access
- But only store data, no support for arbitrary serializable objects (e.g. histograms)

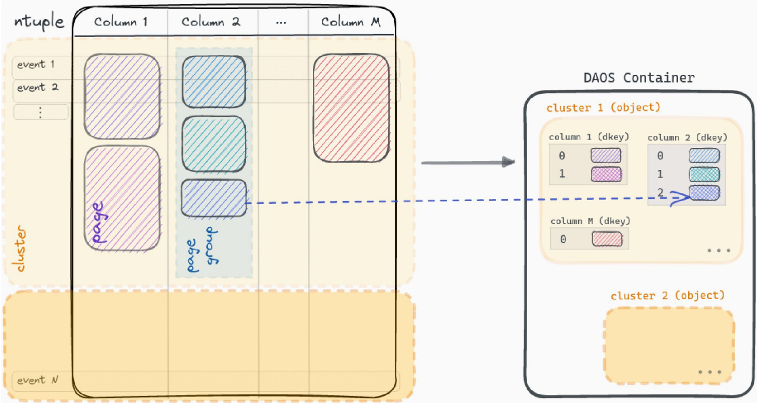
Mapping data to objects

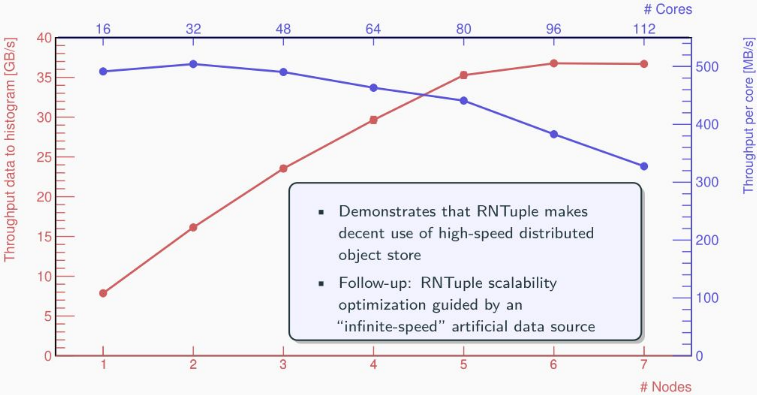
Some deciding factors:

- Granularity: cluster, page…
- Throughput latency
- Cost per request



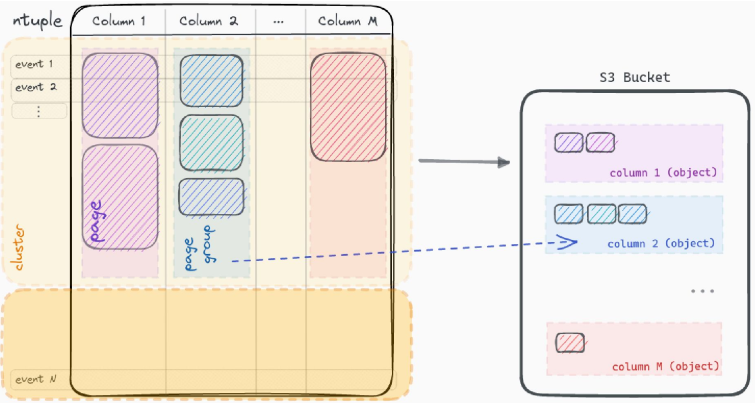
- Foundation for Intel exascale software stack
- Low-latency, high-bandwidth, high IOPS
- Used in top-ranking IO500 systems (e.g. ANL Aurora)
- Native support in RNTuple, demonstrated scaling across multiple DAOS clients

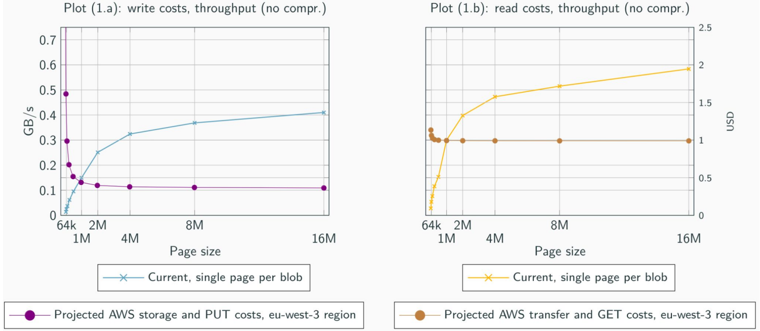




Amazon S3

- De-facto standard object store in Cloud applications
- Different use case w.r.t. DAOS
  - world-wide distributed storage (regions, edges)
  - Network latency becomes noticeable
- First implementation maps columns to S3 buckets
  - Larger objects may mitigate latency





G. Miotto CHEP'23

Conclusions and outlook

- RNTuple towards first production version
  - Clear deliverables set together with experiments
  - Exposure to outside reviewers
- Enable next-generation storage requirements
  - Object stores widely available in cloud environments and some HPC
  - Analysis will benefit from this tight integration too
- Outlook
  - PB scale testing writing and reading RNTuple data
  - Finalizing support for v1 experiment requirements
  - Highly parallel writing