RNTuple: Status and Plans

Jakob Blomer, Florine de Geus, Vincenzo Padulano

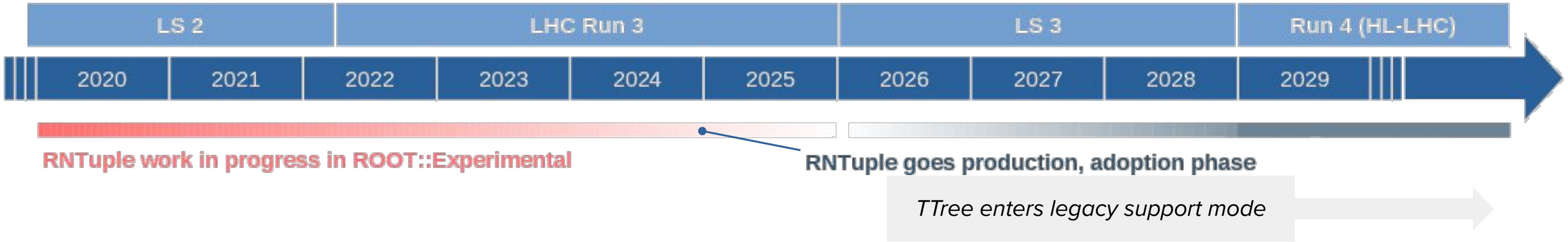
RNTuple Format and Feature Assessment
2023-11-06

ROOT
Data Analysis Framework

https://root.cern

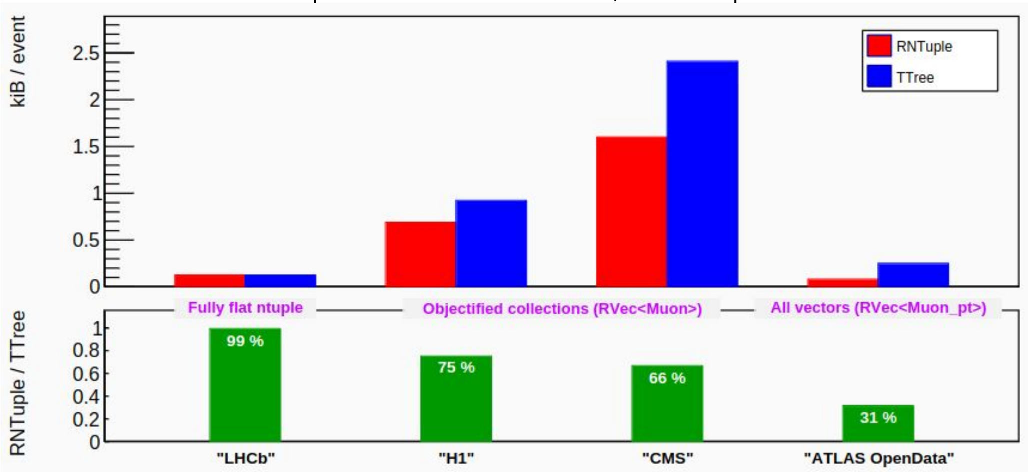
Based on 25+ years of TTree experience, RNTuple is a redesigned columnar I/O subsystem aiming at

- Less disk and CPU usage
  - Significantly **smaller files**
  - Significantly **higher throughput**, often by factors
- Systematic use of **data checksums** and runtime **exceptions** to prevent silent I/O errors
- Efficient support of **modern hardware**:
  - asynchronous & parallel I/O
  - many-core friendly
  - Direct data transfer to GPU memory
- Native support for **object stores** in addition to local and remote ROOT files
- Coverage of all of today's TTree use cases (reconstruction, AOD production, analysis), but not all of the TTree features
- Binary format defined in a dedicated specification

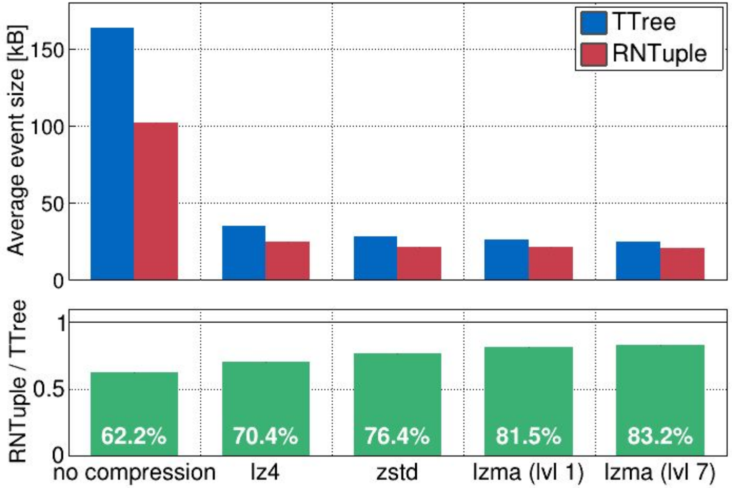


RNTuple Storage Efficiency

RNTuple Standard Benchmarks, **zstd** compression



ATLAS DAOD_PHYS Data Sample (~200k events)

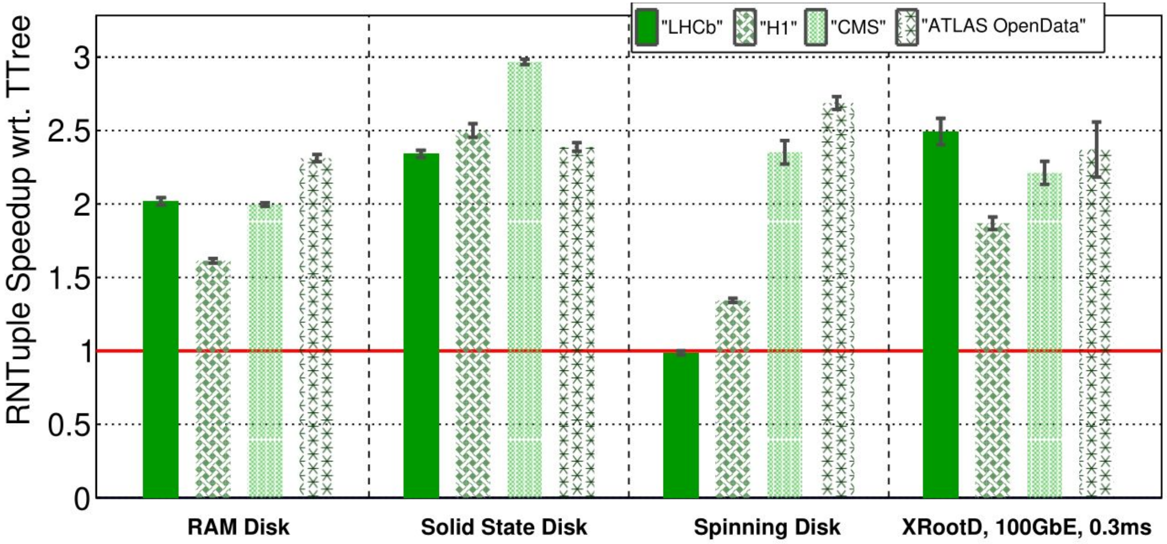


Contributors to space savings

- More **compact representation** of collections and `bools`
- **Data encoding optimized** for better compression ratio

Single-Core RDataFrame Throughput

Time to plot: RDataFrame analysis with RNTuple input data

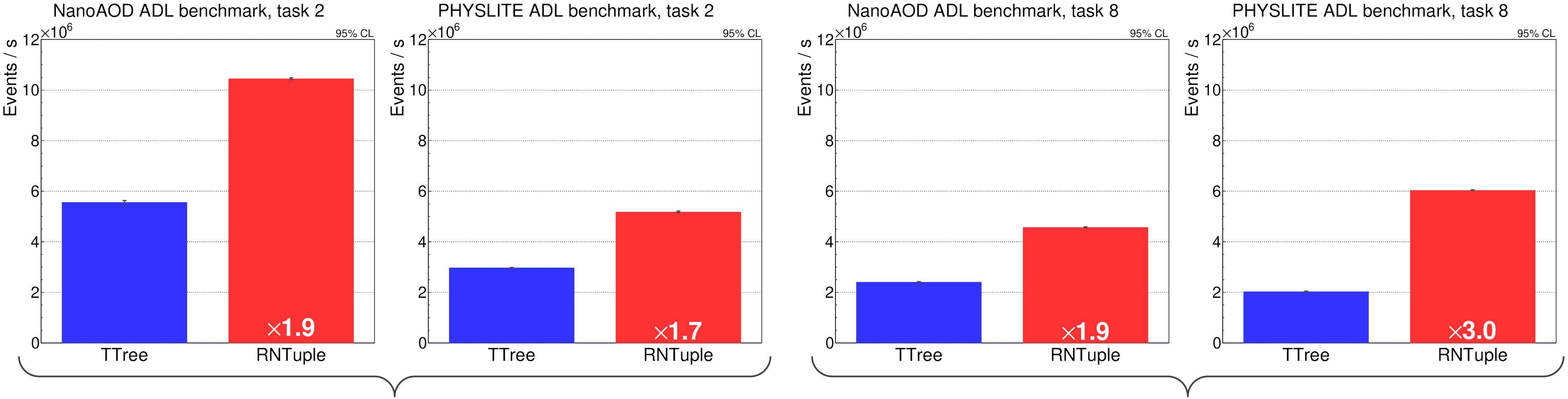


[RNTuple standard benchmarks,](#) input data from various origins

Contributors to higher throughput:

- **Fewer bytes** to read and decompress due to more compact data representation
- **Asynchronous reading**
- **Parallel I/O** improves SSD throughput (uses `io_uring`)
- **Fewer instructions** in the I/O code path

Analysis Description Language (ADL) Benchmarks



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

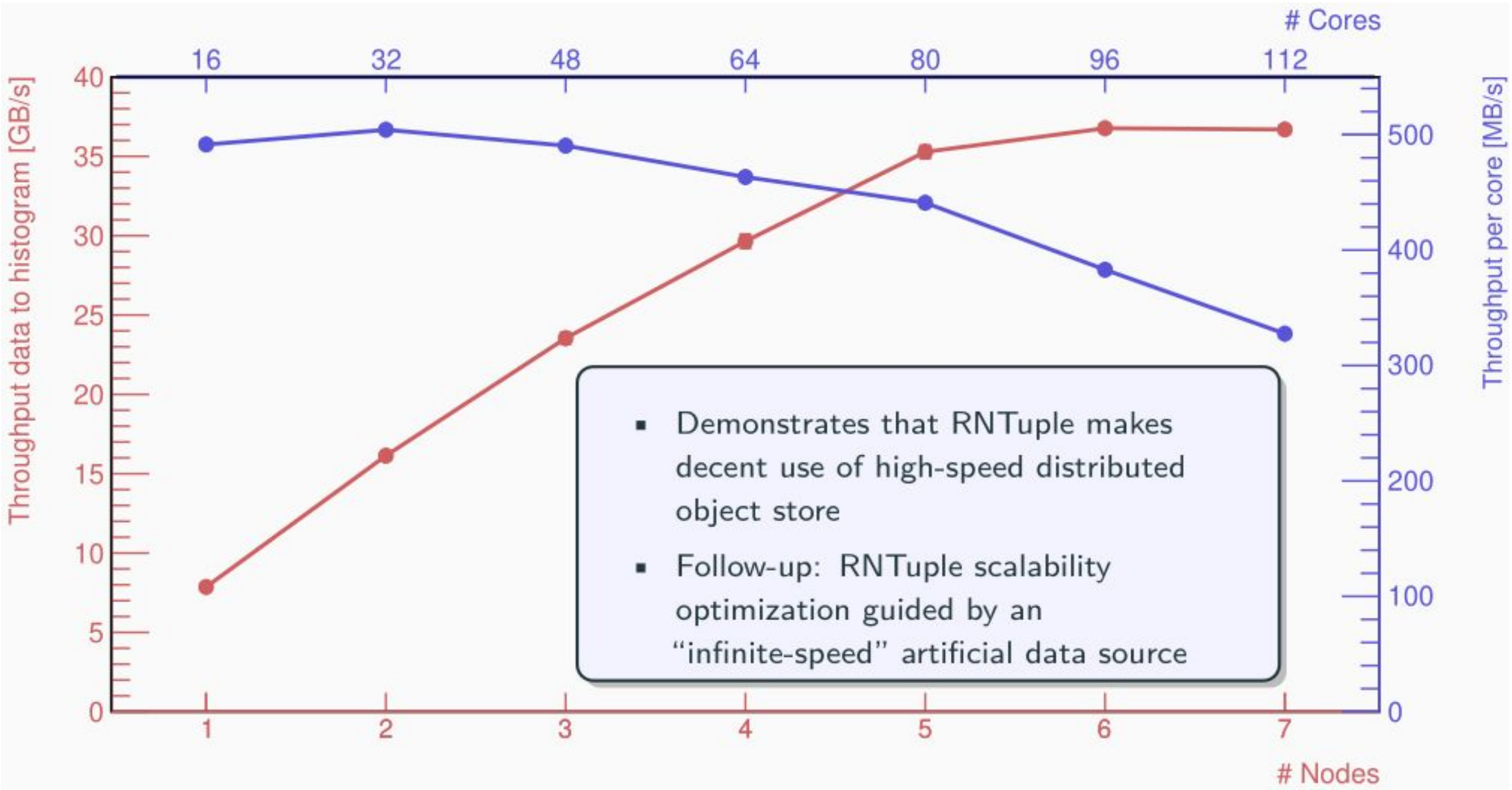
14 (NanoAOD), 12 (PHYSLITE) branches/top-level fields read, some (basic) cuts/calculations applied

- **NanoAOD**: *Run 1* data , 404 MB (TTree), 254 MB (RNTuple), 1.18M events, 86 branches/top-level fields
  - Flat: only scalars/simple collections (stored in leaf count arrays)
- **PHYSLITE**: *Run 2* data, 7.2 GB (TTree), 4.2 GB (RNTuple), 1.18M events, 785 branches/top-level fields
  - xAOD physics objects (not read here) and analysis objects (`std::vector` types)
- Both samples are **`zstd`** compressed
- Single-core throughput with RDataFrame

RDataFrame ADL implementation
for NanoAOD + PHYSLITE

Support for Distributed Object Stores

Distributed RDataFrame on 1TB LHCb ntuples in a DAOS distributed object store, 100Gbit/s network



Potential Future RNTuple Directions

The RNTuple design opens the door to new functionality, which can be worked on after the initial production release.

For example:

- **Horizontal fast merge** ("persistified friends")
- **Zero-copy merge** on copy-on-write file systems
- Better **metadata support** (e.g. scale factors, varied columns)
- **Layout optimizer** that rewrites a file for strictly linear reads

TTree → RNTuple: Overview

RNTuple Compatibility Overview

For maximum optimization opportunities, RNTuple introduces a **new event data format** and a **new API**.
At the same time, RNTuple aims at **smooth integration** with the ROOT/HEP ecosystem.

- RNTuple data stored in **ROOT files**
  - Usual access options: local, XRootD, HTTP
  - **New**: native object store support (DAOS, S3)
- For **RDataFrame** code: no change required
- Consistent tooling
  - **RBrowser** support
  - **Disk-to-disk importer** TTree ➔ RNTuple [1] [2]
  - T[File|Buffer|MPI]Merger & hadd support (RNTuple version 1)
- RNTuple adopts TTree’s I/O customization and schema evolution system (RNTuple version 1)
- Native RNTuple API for writing and reading, targeting frameworks
  - Follows modern C++ core guidelines

- `TTree::Draw` will not be replicated directly in RNTuple;
  a possible replacement on top of RDataFrame is under discussion

```
root [1] .ls
TFile**         /data/gg_data.root
 TFile*         /data/gg_data.root
  KEY: TTree    mini;55 mini [current cycle]
  KEY: TTree    mini;54 mini [backup cycle]
  KEY: ROOT::Experimental::RNTuple      mini_imported;
```

A TTree and an RNTuple in the same ROOT file. In this example, the RNTuple data has been converted from the tree using the `RNTupleImporter`.

Writing and Deriving Data

Entry-by-entry writing:

- Includes "**late model extensions**" to accommodate for frameworks' on-demand schema definition
- Bulk writing: work in progress (bulk reading available)
- Multi-threaded writes:
  - Available: thread-parallel preparation of entries, filling still a serialization point
  - Planned: thread-parallel serialization of complete clusters, *including* writing
    - Work will start in 2024, may land in RNTuple 1.0

Reshaping data: dataset derivation without decompressing / deserialization:

- Fast merging of files, discarding columns (fast "CloneTree")
- Will be available for RNTuple 1.0

Data combinatorics – virtual data sets

- Aligned friends (available) and chains (will be available for RNTuple 1.0)
- EP R&D program on more advanced use cases, such as stored filters, indexed joins, and provenance meta-data; this is considered a potential extension after the first production release (post version 1.0)

Early Adoption

- ATLAS: Experimental support for writing and reading DAOD, AOD, HITS
- CMS: Experimental support for writing NanoAOD files, work-in-progress on MiniAOD
- uproot: Independent implementation of the RNTuple file format; validated the RNTuple format specification

Planned: libRNTupleLite

- Low-level C API to support languages other than C++ and Python (e.g., Julia, Rust)
- Part of a regular ROOT build, i.e. full functionality in principle available and only potentially limited by the C interface
- Depends on 3rd party funding, can be postponed after the first RNTuple production release

RNTuple Type Support

RNTuple supports a subset of the ROOT I/O enabled types

Type Class	Types	EDM Coverage	RNTuple Status
PoD	`bool, std::byte, (unsigned) char, (u)int[8,16,32,64]_t, float, double, (f16)`	Flat n-tuple	Available
(Nested) vectors	`std::vector, RVec, std::array,` 1D C-style fixed-size arrays	Reduced AOD	Available
String	`std::string`		Available
User-defined classes	Non-cyclic classes with dictionaries	Full AOD / ESD / RECO	Available
User-defined enums	Scoped/unscoped enums with dictionaries		Available
User-defined collections	Non-associative collection proxy		Available
`stdlib` types	`std::pair, std::tuple, std::bitset, std::set, std::map, std::atomic`		Available
Alternating types	`std::variant, std::unique_ptr, std::optional` (upcoming)		Available
Intra-event links	`"&Electrons[7]"`		post version 1.0
Low-precision floating points	`Double32_t, Float16_t, (b)float16`	*Optimization benefitting all EDMs*	Avail. / PR
	Custom precision and range		In design / v1.0
	Precision cascades		post version 1.0

RNTuple: Binary Format

RNTuple Binary Format Walk-Through

Benefits of new binary format

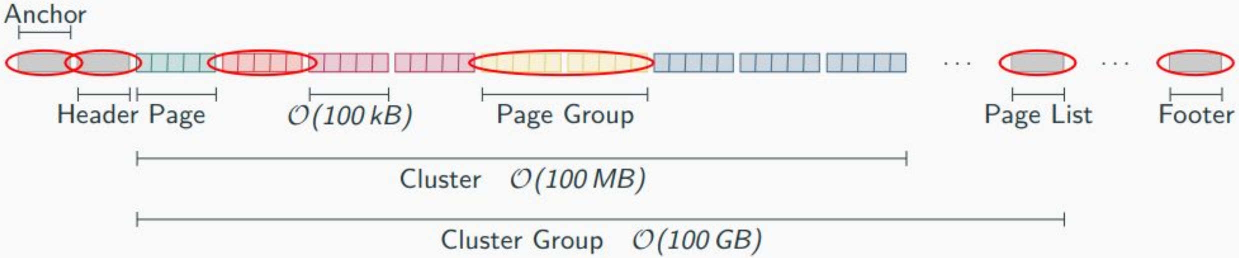
- More efficient storage of collections and boolean values
- Addition of new basic types, e.g. `f16`
- Little-endian numbers: memory mappable on most contemporary platforms
- Type-based encoding: e.g. zig-zag for signed ints, bit packing for bools, etc.
- Split storage for arbitrarily nested collections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

Note: RNTuple has its own schema encoding, independent of the streamer info

```
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 Read Pattern

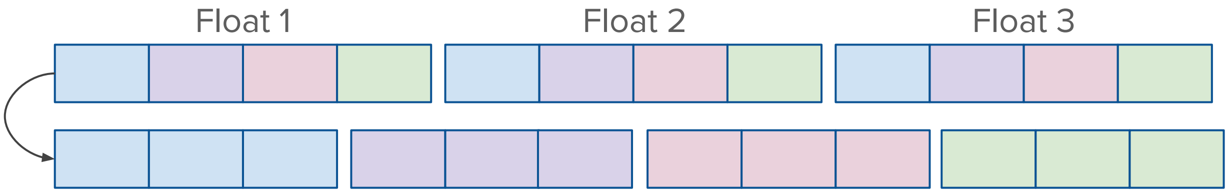
```
struct Event {
  int fId;
  vector<Particle> fPtcls;
};
struct Particle {
  float fE;
  vector<int> fIds;
};
```



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 Column Encodings

- (Simple) transformations on the input data to make them better suited for the compression algorithm, sometimes also called **compression filters**
- Extra computational effort in the noise of the compression algorithm (zstd, etc.)
- Encodings in RNTuple
  - Bools (byte) ➜ bits
  - Floats, Ints ➜ byte split



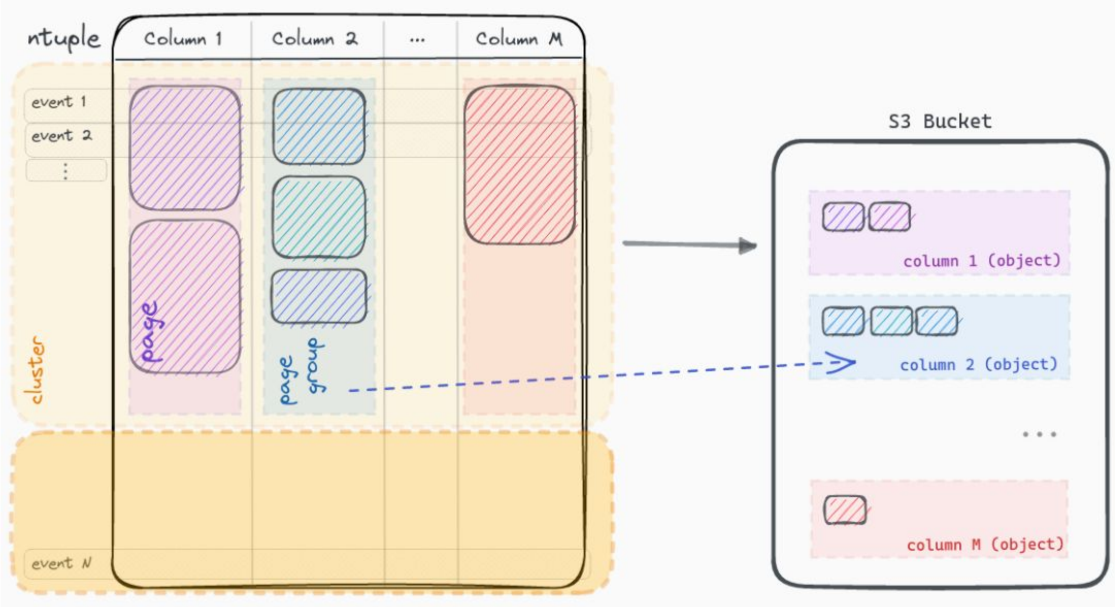
helps if exponent is identical, if least significant bits are zero, and for small ints

  - Collection offset ➜ delta encoding + byte split:
    converts monotonically increasing ints into small, similar/identical ints
  - Signed integers (e.g., charge): zigzag encoding + byte split

$$x \to \begin{cases} 2x, & x \geq 0 \\ -2x - 1, & x < 0 \end{cases}$$

Support for Distributed Object Stores

- RNTuple makes a conscious decision how to map its data structures to objects
- Well suited for columnar analysis approaches
- Currently supported
  - DAOS (HPC)
  - S3 (upcoming, Cloud)
- Investigating reshaping of data during staging from grid storage to object store



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 API

RNTuple Class Design

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`

- General design guidelines
  - Following C++ core guidelines
  - Use of exceptions (`RException`)
  - Conditionally thread-safe
  - Compile-time type-safe interfaces and `void *` interfaces
  - Shared pointers for values to be (de-)serialized
  - 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
  - Can create an `RPageSource`

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

- `RPageSource / RPageSink`
  - Reads and writes pages from the storage backend (file, object store, etc)
  - No concept of entries, only columns
  - Should not be user-facing
  - Gives access to the RNTupleDescriptor

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

- `RNTupleDescriptor`
  - Gives access to the on-disk meta-data

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

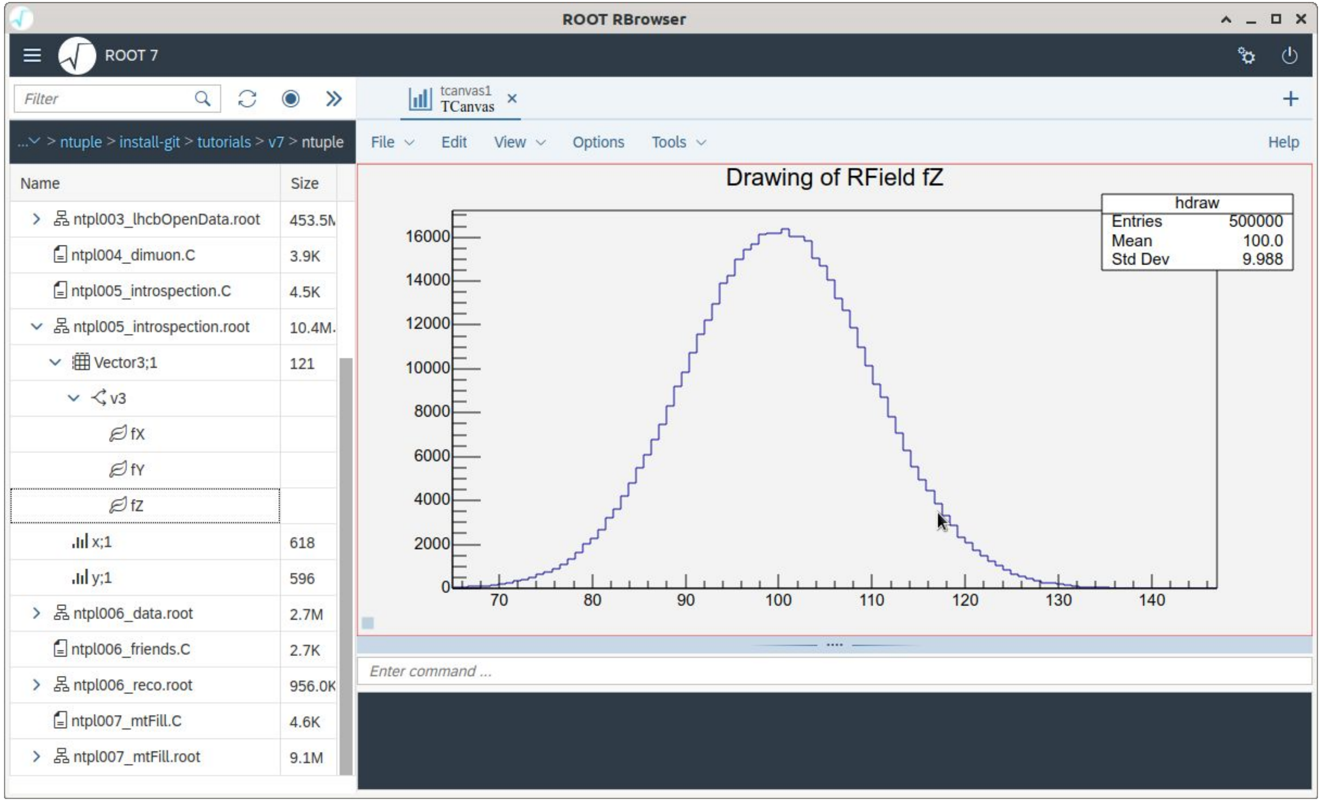
```
auto fieldEta =
  std::make_unique<RField<std::vector<double>>>("eta");
auto fieldPt =
  Detail::RFieldBase::Create("pt", "std::vector<double>").Unwrap();

auto fldDbl32 = dynamic_cast<RField<double> *>(
  std::addressof(*fieldPt->begin()));
fldDbl32->SetDouble32();
```

```
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->CreateBareEntry().lock(); // weak_ptr
  entry->BindRaw("eta", myEta);
  entry->BindRaw("pt", myPt);
  writer->Fill(*entry);
}
```

RNTuple Tooling

RNTuple Data in the RBrowser



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
(CLI coming soon!)

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
(tutorial coming soon!)

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
```

RNTuple Schedule

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

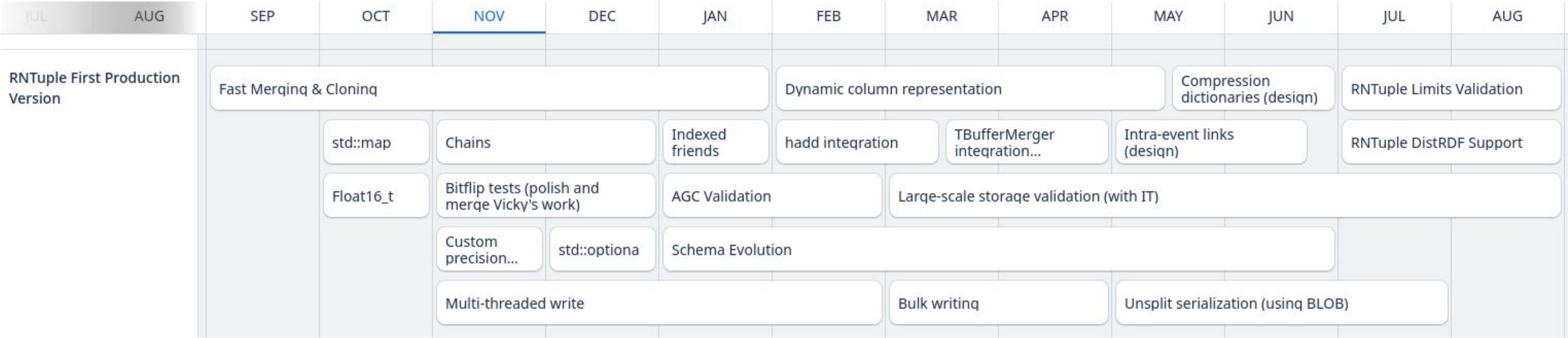
- ☐ PB scale tests
- ☐ Automatic optimization features
- ⛅ Low-precision floats
- ☐ ML Training: direct GPU transfer
- ☐ End-user training
- ⛅ Training and support for code and data migration

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

Work items defined: Nov 2021
Development state: Oct 2023

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

RNTuple Development Plan



Features Foreseen for Removal

Feature Discussion: Features Foreseen for Removal

- References across files, i.e. `TRef` and `TBranchRef`
  - Doesn’t scale and **limits parallelization** potential
  - But support for **intra-event** references is planned!

- **Raw** pointers and **networks** of pointers
  - **Hard** to define the **memory ownership** model

- `std::set`, `std::map`
  - Currently supported, but should discuss potential removal
  - Potentially very slow at runtime

Feature Discussion: Features Foreseen for Removal

- **Dynamic polymorphism** in field types
  - Cause for trouble
  - … and too much runtime overhead

```
struct Base{…};
struct D1: public Base{…};
struct D2: public Base{…};
struct Event{
   std::unique_ptr<Base> fAnyDerived;
};
```

- Recursive data structures
  - **Cannot** properly **split** the type

```
struct Event {
   std::vector<std::unique_ptr<Event>> evts;
};
```

- **Circular** in-memory datasets
  - Unclear benefits

```
TTree t{"name", "title"};
t.SetCircular(10000 /*maxEntries*/);
```

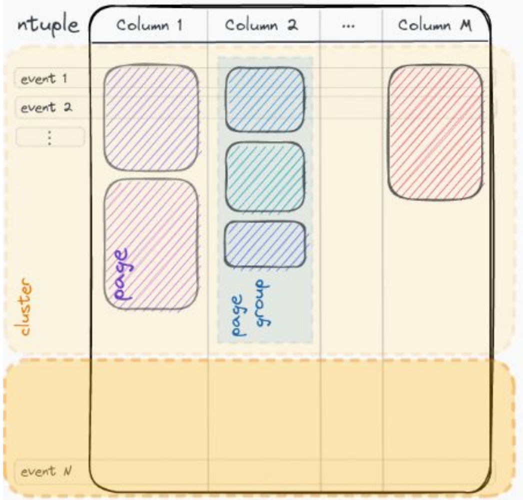
Feature Discussion: Features Foreseen for Removal

- Changing compression on a page level
  - Allow for cluster/column level compression
- TTree::SetAlias
  - And similar APIs that mix processing with data layout
- RNTuple in legacy TBrowser
  - Use RBrowser instead (on by default)

Backup

ROOT Data Compression

- RNTuple: a set of compressed pages/baskets
  - Available compression algorithms:
    - zlib: legacy
    - **zstd**: ROOT 7, RNTuple default
    - lz4: fast, low compression ratio
    - lzma: slow, high compression ratio
  - Arrays of column values are written into pages
  - Compression prefilters (see next slide)
  - Lossless compression is transparent / automatic



- Support for lossy compression
  - **Double32_t**, **Float16_t**: low-precision on disk, double/single precision in memory [1]
    - In addition: allows for fine-grained control over number of mantissa bits
    - Or: range specification with bit resolution, e.g. [0..π, 6 bits]
    - Full functionality set available in TTree, soon also in RNTuple
  - R&D: BLAST compression algorithm & **Precision Cascades (ongoing R&D)**

Precision Cascades I/3

...

2.71934

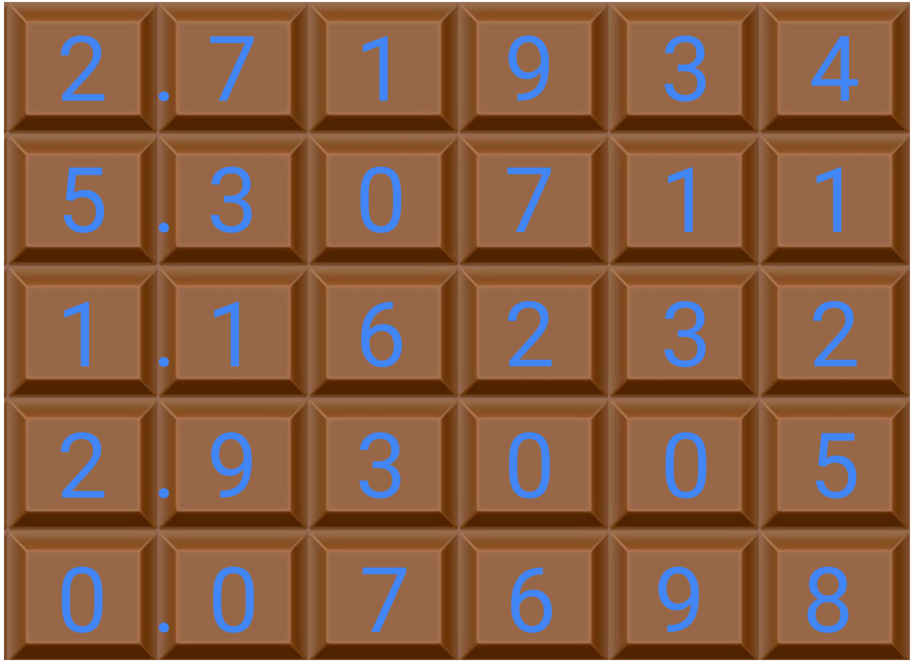
5.30711

1.16232

2.93005

0.07698

...


2.71934
5.30711
1.16232
2.93005
0.07698

Precision Cascades 3/3

Ongoing R&D

- Enables higher precision to be stored separately without duplicating information
- User can define levels of precision
  - Varying levels of precision can be retrieved

```
std::vector<Int_t> levels = { 51, 43 };
ROOT::PrecisionCascadeCompressionConfig targetConfig(
      ROOT::RCompressionSetting::EAlgorithm::kBLAST,
      levels,
      true /* Keep also the residual file */ );
…
lossy_branch->SetCompressionSettings(targetConfig);
```