# UnROOT.jl - Status Update & RNTuple

**JuliaHEP 2024 @CERN**

**Jerry Ling (Harvard University),** Tamás Gál (ECAP)

# What is UnROOT.jl for?

- A Julia Package for reading and (soon™) writing .root files
- Reading:
  - TTree, RNTuple, histograms etc.
- Writing:
  - **RNTuple**

# History

- ❖ In 2021, we have mostly focused on “reading” TTrees and histograms to facilitate end-user analysis workflows.
- ❖ In 2022, improved performance and coverage. “Dogfooding” in our ATLAS analysis (sending to journal soon).
- ❖ Since 2022, working on reading RNTuple, already very wide coverage, feed back into the ROOT team’s R&D process.
- ❖ Since last JuliaHEP, prototyping RNTuple writing

# Structure of this talk

- What's special about  .root files?
- Introduction (or recap) of UnROOT.jl features
- Crash course on RNTuple
- Status of RNTuple I/O

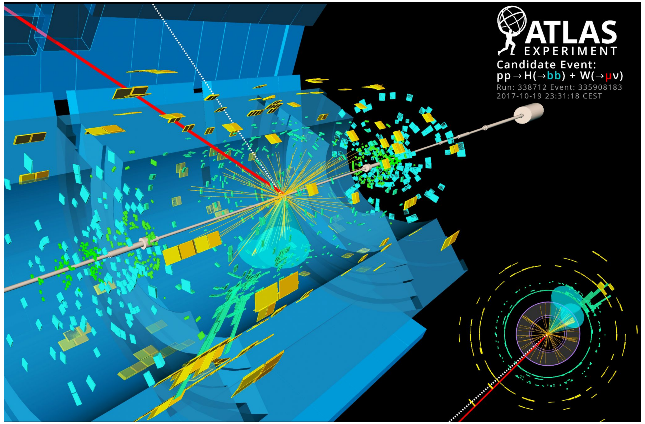
- What's special about  .root files?
- Introduction (or recap) of UnROOT.jl features ⏳
- Crash course on RNTuple ⏳
- Status of RNTuple I/O ⏳

# ROOT and .root files

Two challenges in HEP data:

1. Immense data size (need performance and compression ratio)
2. Complex, hierarchical data model

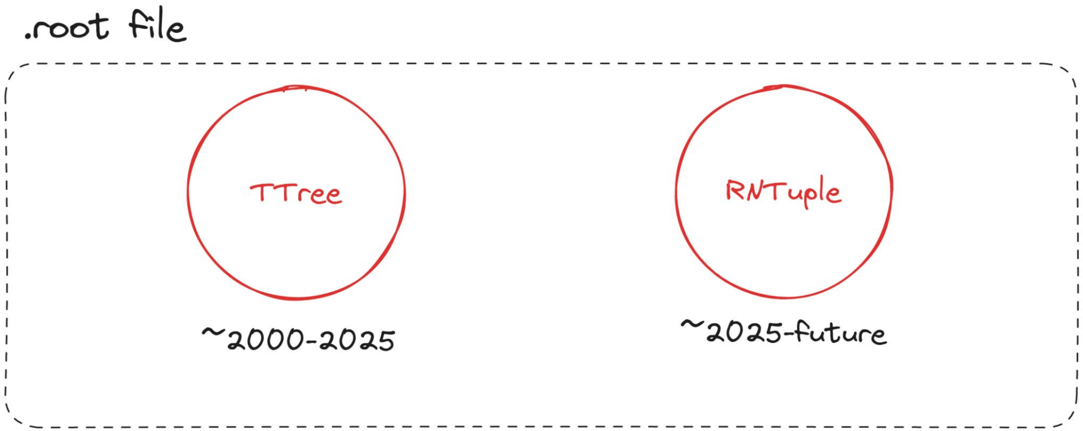
ROOT C++ framework and .root file were created at CERN to deal with them.




ROOT
Data Analysis Framework

# TTree -> RNTuple

- ❖ In short, the complex data structure pushed HEP to invent its own data format: TTree and RNTuple
- ❖ They share only a few things in design, with RNTuple posed to replace TTree completely




*Two table-like objects in .root files used for physics events data*

- What's special about .root files? ✅
- Introduction (or recap) of UnROOT.jl features
- Crash course on RNTuple ⏳
- Status of RNTuple I/O ⏳

# Existing UnROOT.jl features:

❖ Tables.jl-compatible representation of TTrees / RNTuples

```
julia> mytree = LazyTree(f, "Events", ["Electron_dxy", "nMuon", r"Muon_(pt|eta)$"])
 Row │ Electron_dxy                      nMuon   Muon_pt          Muon_eta
     │ SubArray{Float3                   UInt32  SubArray{Float3  SubArray{Float3
─────┼──────────────────────────────────────────────────────────────────────────────
 1   │ [0.000371]                        0       []               []
 2   │ [-0.00982]                        2       [19.9, 15.3]     [0.53, 0.229]
 3   │ []                                0       []               []
 4   │ [-0.00157]                        0       []               []
 5   │ []                                0       []               []
 6   │ [-0.00126]                        0       []               []
 7   │ [0.0612, 0.000642]                2       [22.2, 4.43]     [-1.13, 1.98]
 8   │ [0.00587, 0.000549, -0.00617]     0       []               []
  ⋮  │              ⋮                     ⋮             ⋮                ⋮
                                                                992 rows omitted
```

# Existing UnROOT.jl features:

❖ Lazy I/O during event iteration of wide table

events::LazyTree

```
evt = events[1] # no I/O happens
evt.Elec_qualities # I/O happens here
```

# Existing UnROOT.jl features:

❖ Transparently thread-safe

```
for evt in events
    for e in evt.Elec_4vector
        if e.pt > 10.0
            push!(hist_elec_eta, e.eta)
        end
    end
end
```

```
@threads for evt in events
    for e in evt.Elec_4vector
        if e.pt > 10.0
            atomic_push!(hist_elec_eta, e.eta)
        end
    end
end
```

# Performance techniques for reading

Three most important things for reading in general:

- ❖ Type stability
- ❖ Lazy data materialization
- ❖ Chunked caching

# Performance #1: type stability

- For this to be fast, compiler must be able to infer the type of \`evt.Elec_4vector\` and so on.
- Solution: Encode name <-> type mapping in the type info of \`evt\`.

```
for evt in events
    for e in evt.Elec_4vector
        if e.pt > 10.0
            push!(hist_elec_eta, e.eta)
        end
    end
end
```

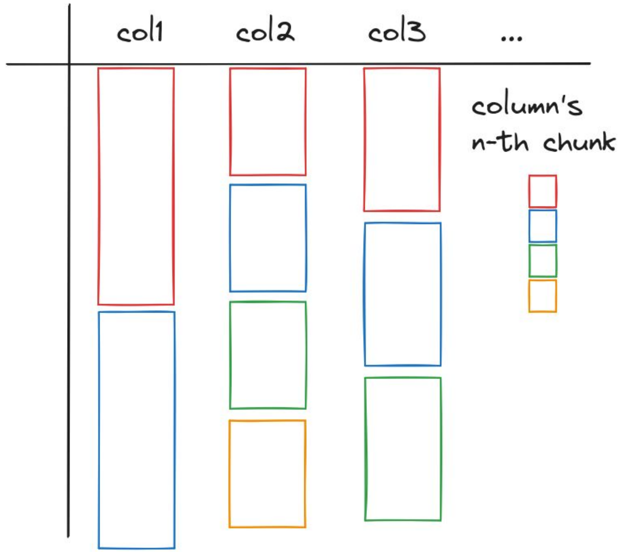
# Performance #2: lazy materialization

- Often, the `events` may have O(1000) columns, but users may only access O(10)
- Solution: Delay the reading of column content until something like `evt.Col1` actually happens.

```
evt = events[1] # no I/O happens
evt.Elec_qualities # I/O happens here
```

# Performance #3: chunked cache

- Both TTree and RNTuple are “columnar”, meaning multiple rows of the same column are stored together on disk.
- When reading “1 row”, you are forced to do the work for many (1k-100k) rows.
- Solution: cache the chunk and its range, per column.




*Chunk spans multiple rows*

# Performance #3.5: chunked cache with thread-safety

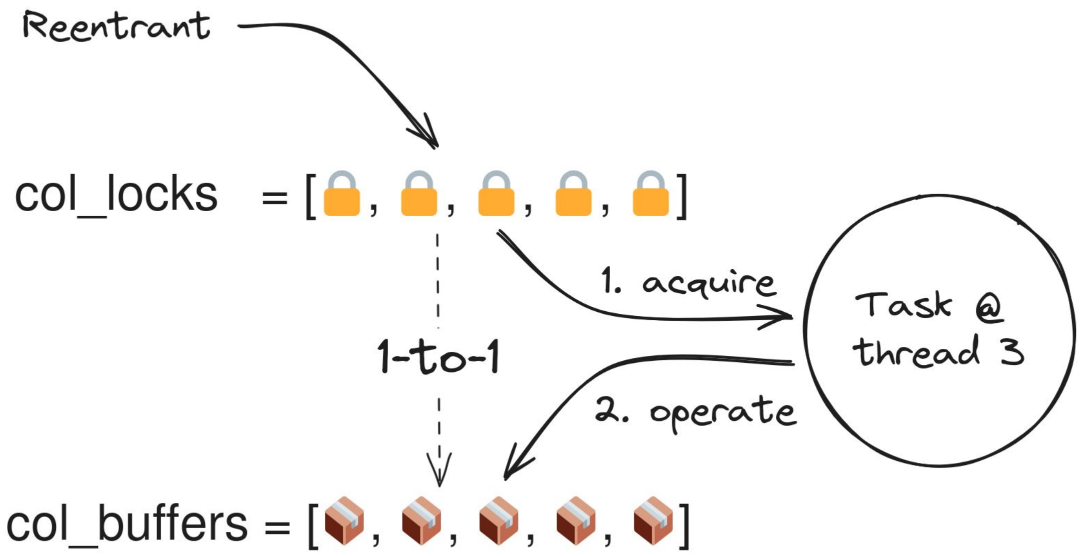
- To make the chunk caches thread-safe, you need a cache per column, per thread.
- Initially, it was done with `buffers[threadid()]`, then I asked about it in Julia slack, long discussion ensued.
- Result:

**PSA: Thread-local state is no longer recommended**

06 July 2023 | **Mason Protter, Valentin Churavy, Ian Butterworth, and many helpful contributors**

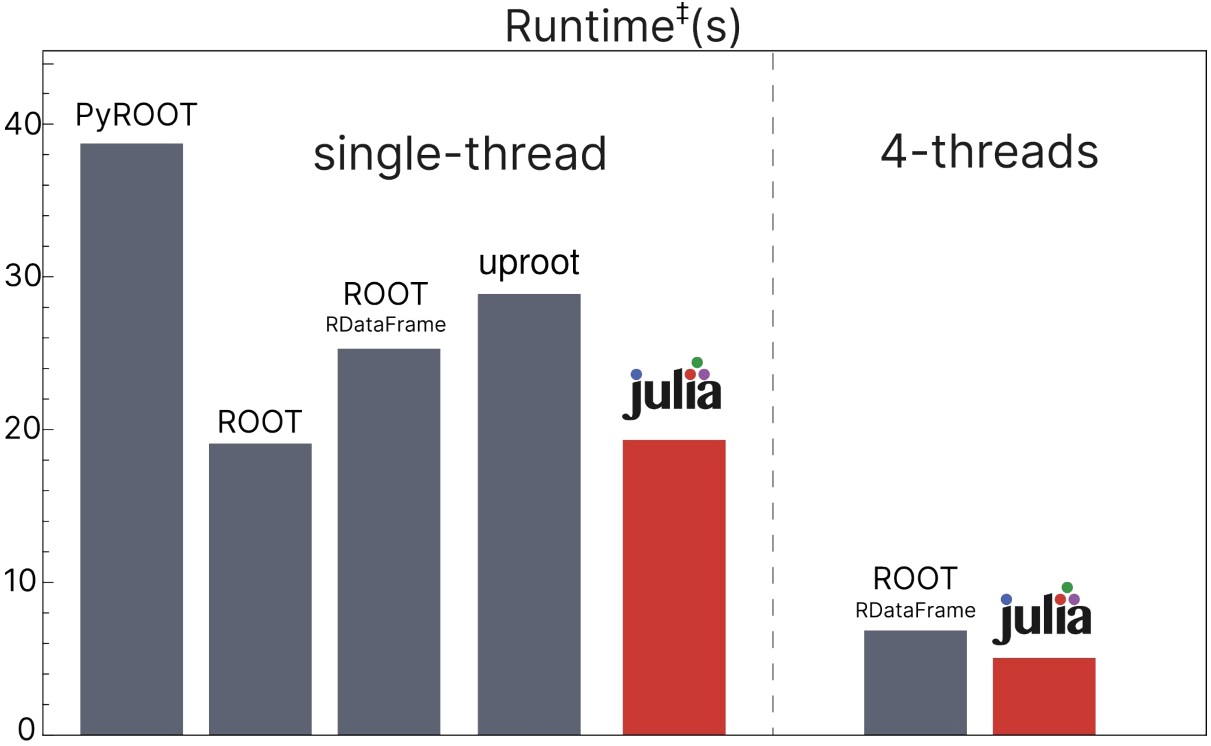
# Performance #3.5: chunked cache with thread-safety

- New strategy:




- Now it is safe even when a task migrates to the thread where another task is running.

# TTree reading performance:



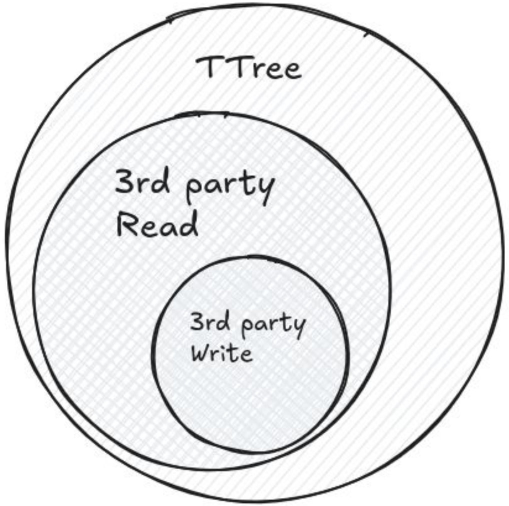

‡: Exact ranking depends on the workload

cern.ch/go/vhR6

- ❖ What’s special about .root files? ✅
- ❖ Introduction (or recap) of UnROOT.jl features ✅
- ❖ Crash course on RNTuple
- ❖ Status of RNTuple I/O ⏳

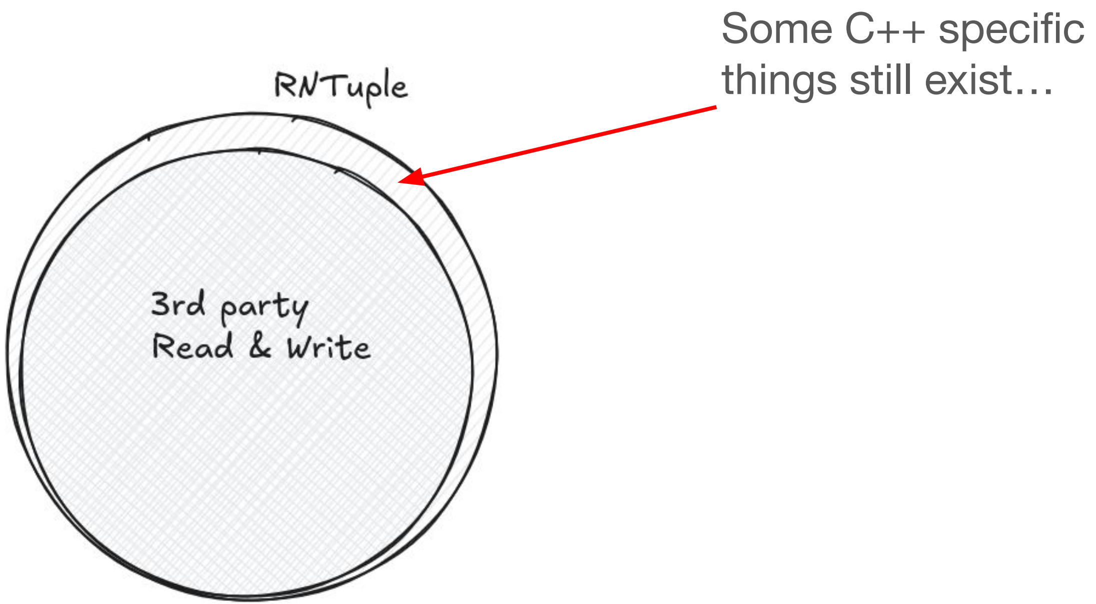
# What is RNTuple

- One drawback of TTree is the lack of “specification” – which created a messy compatibility landscape:



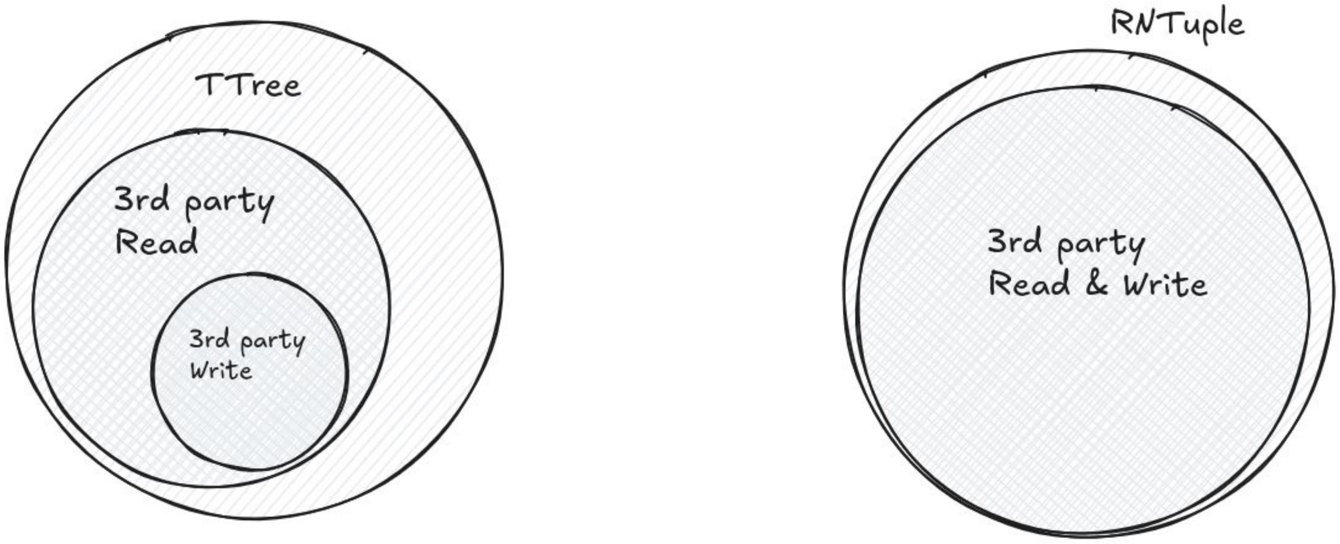
# What is RNTuple

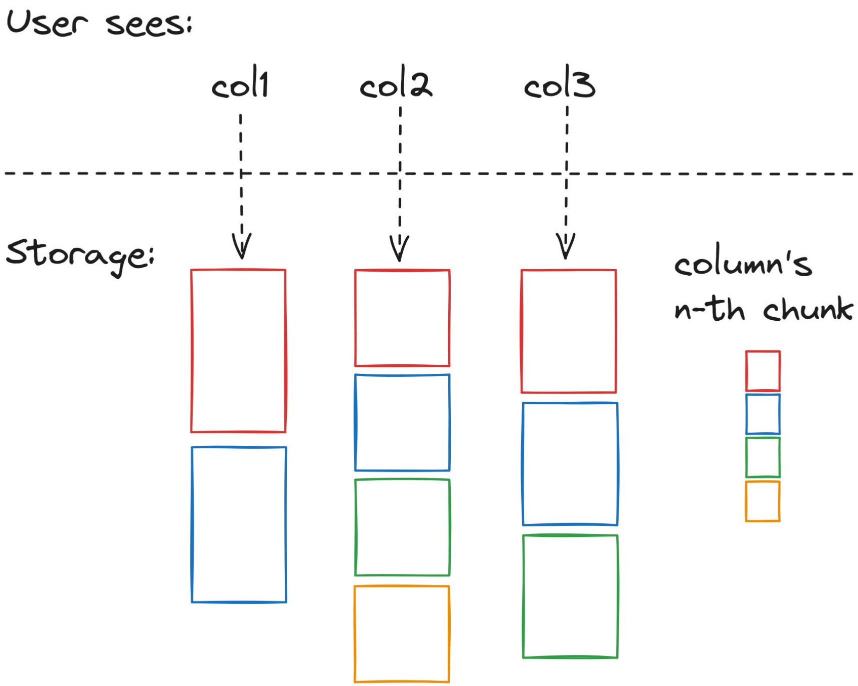
- In RNTuple, we can expect much more uniform compatibility thanks to specification-oriented design:



# What is RNTuple

- It is helpful to draw contrasts between TTree and RNTuple in order to explain why RNTuple's design is more "principled"

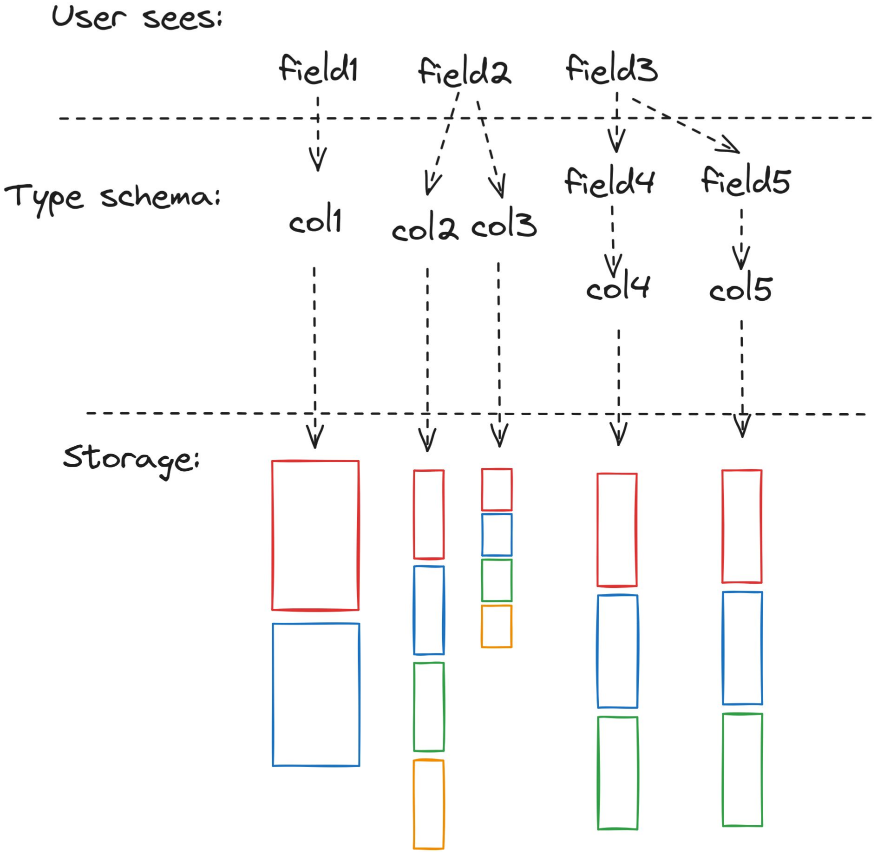





In **TTree**, every column the user sees correspond to one group of storage units.

If `col` is complex: squeeze heterogeneous data into the same storage unit -> bad compression.

- **RNTuple's** design is more similar to Apache Parquet/Arrow(Feather):




In **RNTuple**, every column user sees can be composition of fields/columns.

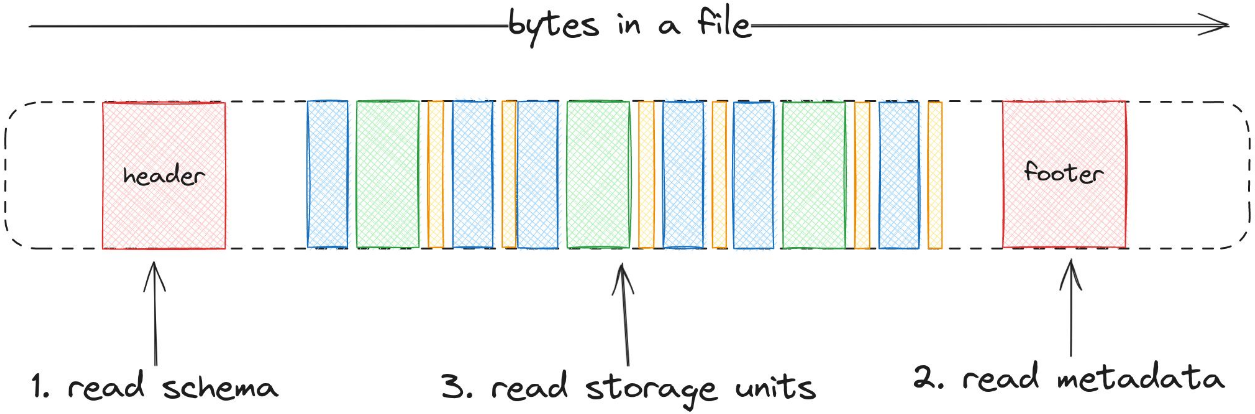
This allows better compression efficiency and uniform schema composition rule.

- What's special about .root files? ✅
- Introduction (or recap) of UnROOT.jl features ✅
- Crash course on RNTuple ✅
- Status and update on RNTuple I/O

# RNTuple reading strategy:

Reading of the columnar format can be broken down into 3 steps:

1. Parse metadata for type schema
2. Process referential metadata (i.e. where are the storage units)
3. Compose Julia types and attach storage units accordingly.



# RNTuple reading strategy: type schema

❖ Through extensive use of multiple-dispatch, manipulation in **type space** is more modular and less error-prone when containers nest each other.

```julia
struct UnionField{S,T}
    switch_col::S
    content_cols::T
end
isvoid(::Type{<:UnionField}) = false

function _parse_field(field_id, field_records, column_records, alias_
    switch_col = _search_col_type(field_id, column_records, alias_col
    element_ids = findall(field_records) do field
        field.parent_field_id == field_id
    end
```

```julia
# the parent field is only structral, no column attached
struct StructField{N, T}
    content_cols::T
end
function isvoid(::Type{StructField{N,T}}) where {N,T}
    isvoid(T) #|| all(startswith(":_"), String.(N))
end

function _parse_field(field_id, field_records, column_records,
    element_ids = findall(field_records) do field
        field.parent_field_id == field_id
```

# In real-world application, we do push the schema type system very far:

```
├─ Symbol("AntiKt4TruthWZJetsAux:") ⇒ Struct
│                                      ├─ :m ⇒ Vector
│                                      │        ├─ :offset ⇒ Leaf{UnROOT.Index64}(col=165)
│                                      │        └─ :content ⇒ Leaf{Float32}(col=166)
│                                      ├─ :pt ⇒ Vector
│                                      │         ├─ :offset ⇒ Leaf{UnROOT.Index64}(col=159)
│                                      │         └─ :content ⇒ Leaf{Float32}(col=160)
│                                      ├─ :eta ⇒ Vector
│                                      │          ├─ :offset ⇒ Leaf{UnROOT.Index64}(col=161)
│                                      │          └─ :content ⇒ Leaf{Float32}(col=162)
│                                      ├─ :constituentWeights ⇒ Vector
│                                      │                         ├─ :offset ⇒ Leaf{UnROOT.Index64}(col=171)
│                                      │                         └─ :content ⇒ Vector
│                                      │                                       ├─ :offset ⇒ Leaf{UnROOT.Index6
│                                      │                                       └─ :content ⇒ Leaf{Float32}(col
│                                      ├─ :phi ⇒ Vector
│                                      │          ├─ :offset ⇒ Leaf{UnROOT.Index64}(col=163)
│                                      │          └─ :content ⇒ Leaf{Float32}(col=164)
│                                      └─ :constituentLinks ⇒ Vector
│                                                              ├─ :offset ⇒ Leaf{UnROOT.Index64}(col=167)
│                                                              └─ :content ⇒ Vector
│                                                                            ├─ :offset ⇒ Leaf{UnROOT.Index64}
│                                                                            └─ :content ⇒ Struct
│                                                                                          └─ Symbol(":_0") ⇒
```

# RNTuple reading wish list 🚧

- One minor inconvenience for reading is we're not fully efficient when the user only wants to access a sub-field of a structure.
- For example, if the user only uses `**evt.Ak4jets.pt**`, in principle, we only need to touch two columns: one for offset, one for content.
- But our current "lazy" strategy stops when user access `evt.Ak4jets`, we end up reading everything under `Ak4jets` field.

# RNTuple reading wish list 🚧

- The current implementation (which uses StructArrays.jl and ArraysOfArrays.jl) doesn't give us enough control over the whole access.
- One possible approach is to take more control over the whole `LazyTree`, for example, by using AwkwardArray.jl
- This can help us even more in "Writing", see later slides.

# RNTuple is still evolving:

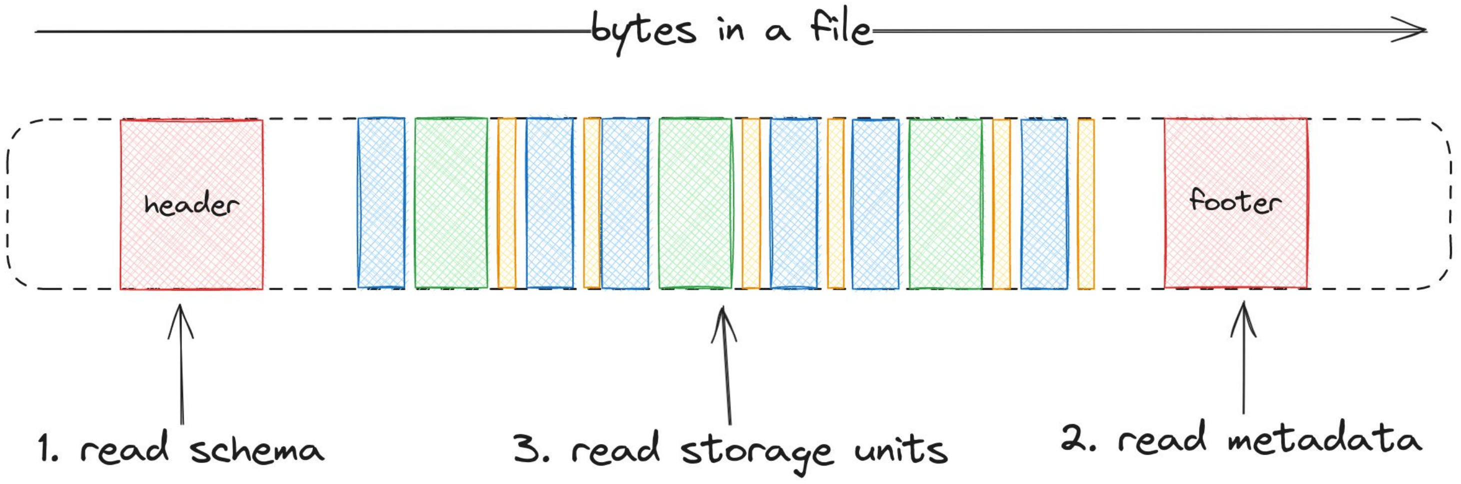
- Before delve into **writing**, note that RNTuple is still having breaking changes from time to time.
- A handful of breaking changes (adding/removing fields from data structure, adding new checksum, changing positive and negative values etc.)
- Expected to freeze around CHEP 2024 (in one month)

Takeaway: do not prematurely optimize our implementation.

# RNTuple writing strategy:

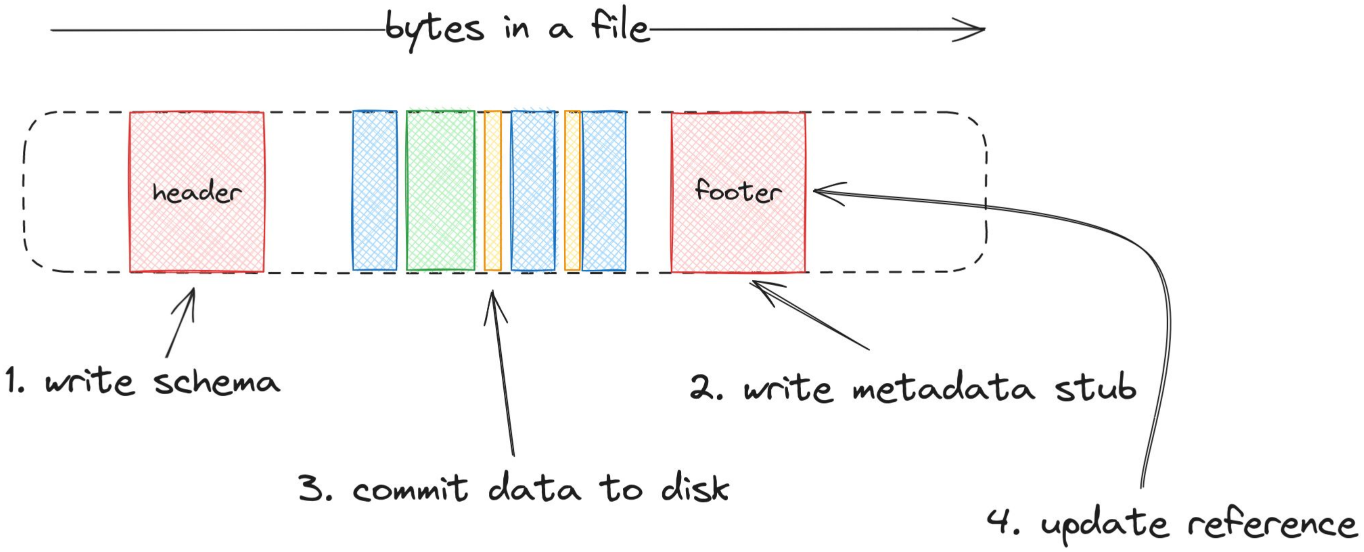
❖ Writing is very different from reading, in fact, almost no code can be reused.

❖ Information flow **during reading**:



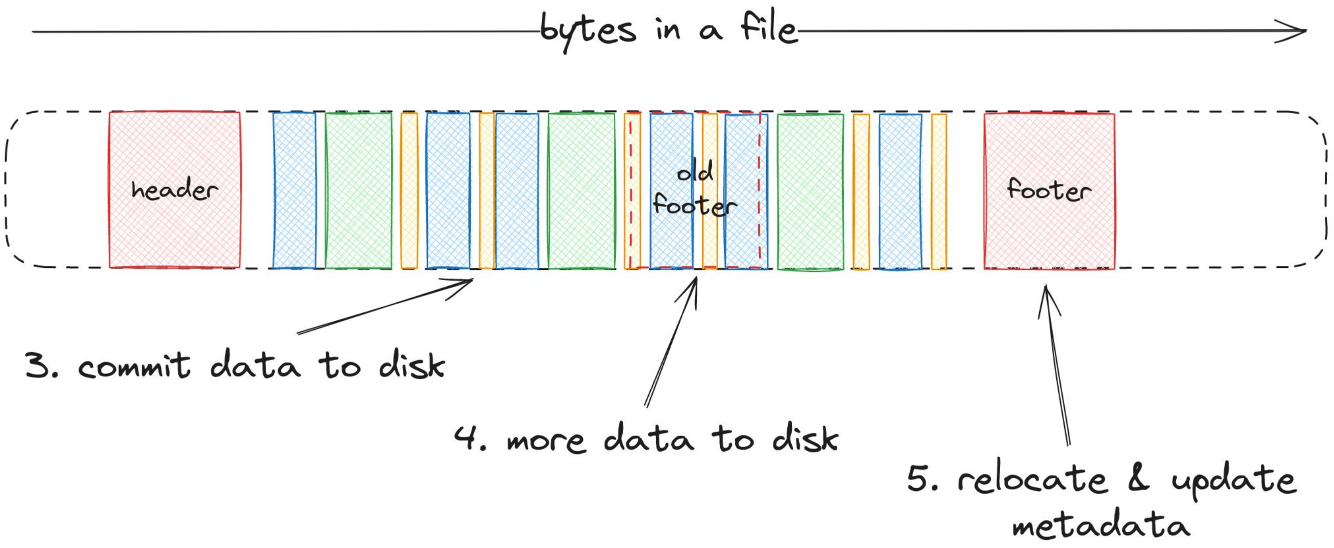
# RNTuple writing strategy:

- **For writing**, you need to alternate between committing storage units to disk and update referential metadata:



# RNTuple writing strategy:

❖ Often, data are too big to write in one go, so relocation of the metadata blocks are needed:



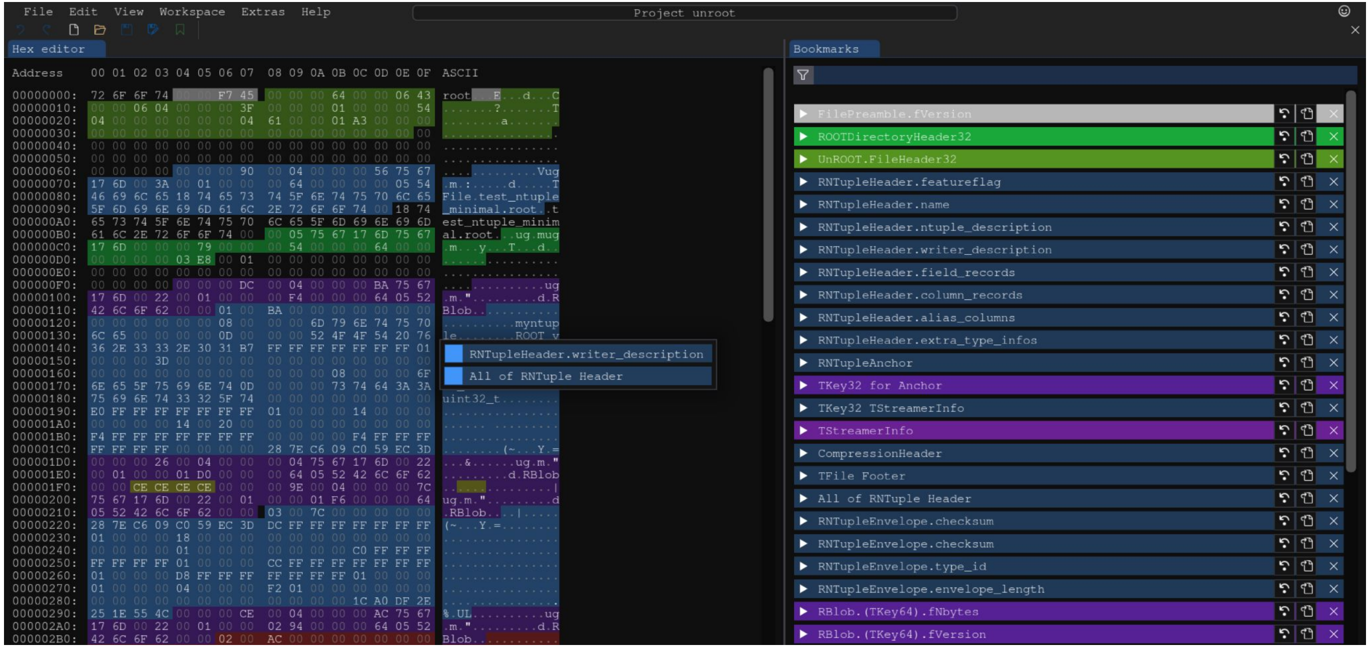
# Development plan:

Breakdown the development into three phases, with incrementing level of completeness and automation:

1. Proof-of-concept: use as much hard-coded byte blobs as needed (#343 in June) ✅
2. Minimally viable for end-user: common types for analysis, large table, compression etc. (#349 now)
3. “Advanced” features: Complex types, efficient appending, streaming etc.

# RNTuple writing: #0

- Although RNTuple has specification, not everything in a .root file is. So the 0th step is to open a hex editor and understand every single byte:

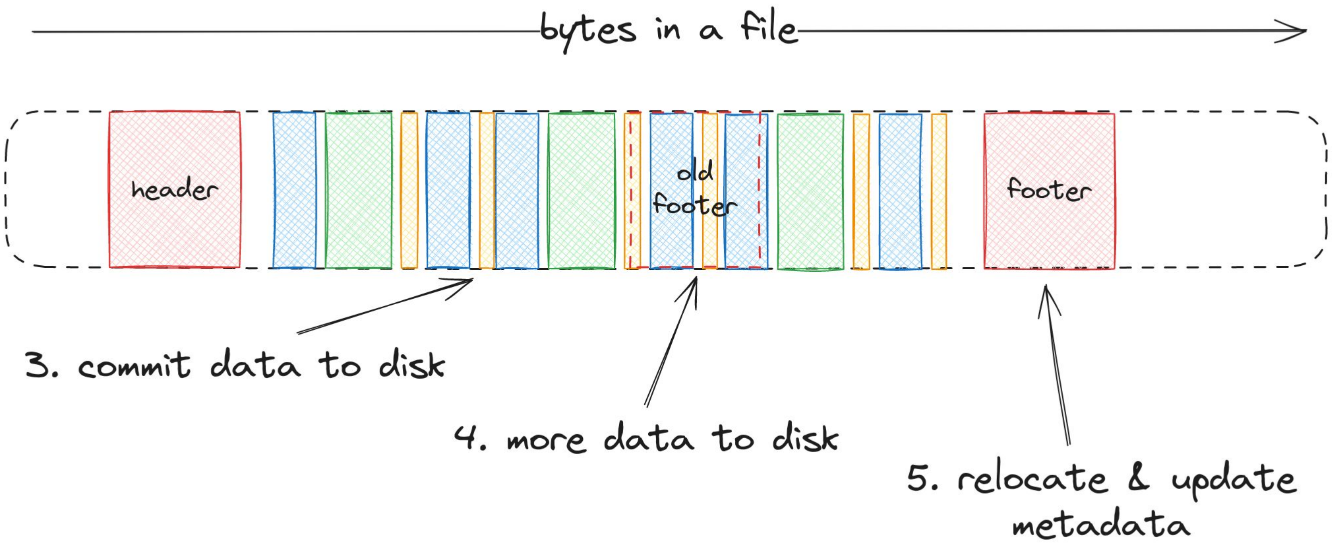


# RNTuple writing: #1

- After understanding every single byte, create stubs for things.
- For file metadata parts without specification, reuse **byte blobs.**
- For the parts that have specification, write **Julia objects** and I/O to re-create them.
- Using a dynamic language helped immensely during this iterative development.

# RNTuple writing: #2

❖ Using Observables.jl-like structure to keep a record on metadata object, when they get updated, flush updated bytes to disk.



# RNTuple writing: Current status

```
julia> data = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
       newtable = (
               x1 = Float64.(data),
               x2 = Float32.(data),
               x3 = Int32.(data),
               y1 = UInt16.(data),
           )
       UnROOT.write_rntuple(open("/tmp/a.root", "w"), newtable; rntuple_name="myntuple")
       LazyTree("/tmp/a.root", "myntuple")
```

| Row | x1 Float64 | x2 Float32 | x3 Int32 | y1 UInt16 |
|---|---|---|---|---|
| 1 | 5.0 | 5.0 | 5 | 5 |
| 2 | 6.0 | 6.0 | 6 | 6 |
| 3 | 7.0 | 7.0 | 7 | 7 |
| 4 | 8.0 | 8.0 | 8 | 8 |
| 5 | 9.0 | 9.0 | 9 | 9 |
| 6 | 10.0 | 10.0 | 10 | 10 |
| 7 | 11.0 | 11.0 | 11 | 11 |
| 8 | 12.0 | 12.0 | 12 | 12 |
| 9 | 13.0 | 13.0 | 13 | 13 |
| 10 | 14.0 | 14.0 | 14 | 14 |

# RNTuple writing road ahead 🚧

The biggest long-term challenge is how to have near-100% coverage of all possible types users want to serialize, two related challenges:

1. Generate (arbitrarily) nested fields and columns schema data
2. Re-organize Julia objects into primitive storage units (offset, content etc.)

A systematic approach can be helpful.

# RNTuple writing road ahead 🚧

AwkwardArray.jl is one of such systematic approaches.

Given a table-like data structure, it will be able to output:

- ❖ A type schema / tree that is compatible with RNTuple (with simple translation for the base unit)
- ❖ An in-memory layout with appropriate basic columns such as "offset" and "content" already transformed.

# Summary

- UnROOT.jl is feature-rich and fast for common end-user analysis applications
- Following RNTuple development and will be ready when the switch happens.
- RNTuple writing is steadily maturing, and integration with AwkwardArray.jl can be an exciting solution towards feature-completeness. See Ianna‘s talk at 11am!

*Hackathon: finish #349, learn RNTuple and reverse engineering ROOT logics!*

# Backup

# RNTuple and reading it from Julia

- RNTuple is the upcoming, brand new format for storing data beginning 2025.
- The design is similar to some industry formats emerged in the last decade:

| RNTuple | Parquet | Arrow/Feather |
|---|---|---|
| field | column | field |
| column | – | array |
| cluster | row group | row group |
| page list | column chunk | record batch |
| page | page | buffer |

*Terminology translation between columnar formats*

# RNTuple reading: type schema

- Through extensive use of multiple-dispatch, manipulation in type-space is more modular and less error-prone when containers nest each other.
- For example, consider a column with eltype “vector of structs”.
- This involve two different containers:
  - Vector
  - Struct

# RNTuple reading: type schema

- The “vector” by itself is encoded using “content and offset” approach:

User sees: `ary = [[12, 14], [], [17, 19, 21]]`

What's actually stored:

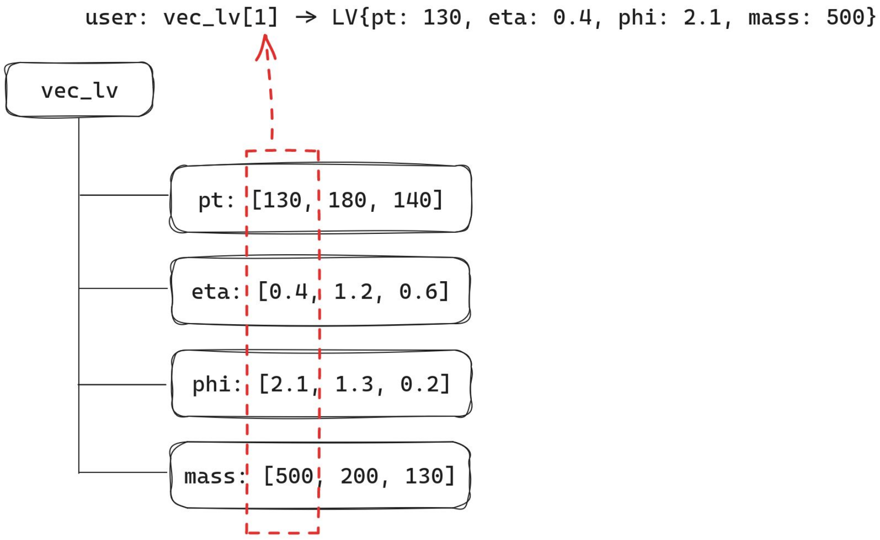
```
content = [12, 14, 17, 19, 21]
offset  = [0, 2, 2, 5]

ary[0] = content[0:2] = [12, 14]
```

*“Content and offset” for jagged vector, similar to ArraysOfArrays.jl*

# RNTuple reading: type schema

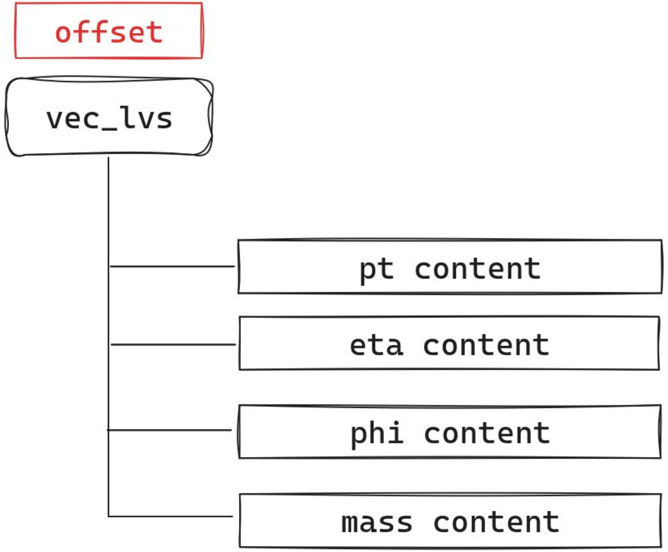
- The “struct” by itself is encoded using “struct of arrays” approach:




*Struct of arrays encoding, similar to StructArrays.jl*

# RNTuple reading: type schema

- The power of the design and our strategy is that they can compose freely:




*Schema of a column with eltype "vector of structs"*