RNTuple Implementation in julia

CHEP 2024 @ Kraków, Poland

Jerry Ling (Harvard University / ATLAS)
Tamás Gál (Erlangen Centre for Astroparticle Physics)

What is Julia?

❖ Luckily, Graeme has given the plenary talk introducing Julia in the context of HEP:

Julia in HEP

21 Oct 2024, 11:00

30m

Large Hall

Speaker

Graeme A Stewart (CERN)

What is RNTuple?

- However, the RNTuple plenary talk is still in the future:

ROOT RNTuple and EOS: 0

23 Oct 2024, 11:00

30m

Large Hall

Speakers

Andreas Joachim Peters (CERN)

Jakob Blomer (CERN)

Structure of this talk

- What's special about RNTuple (short version)
- Implementation highlights in UnROOT.jl
- Current status and exciting future

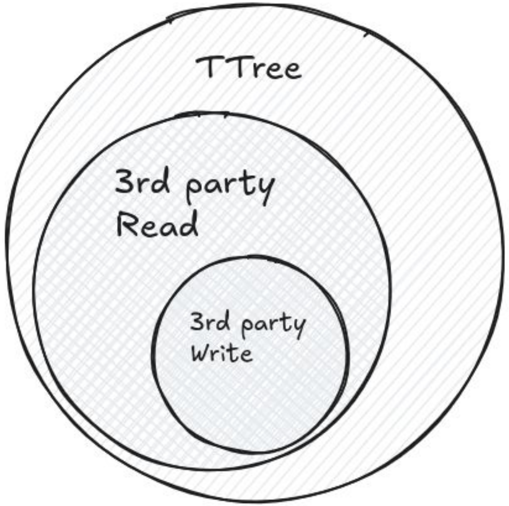
What is RNTuple

- In short, RNTuple is the next-gen evolution of TTree.
- TTree and RNTuple both live inside .root files, but don't share much in their design or implementation.

Both are table-like objects in .root files

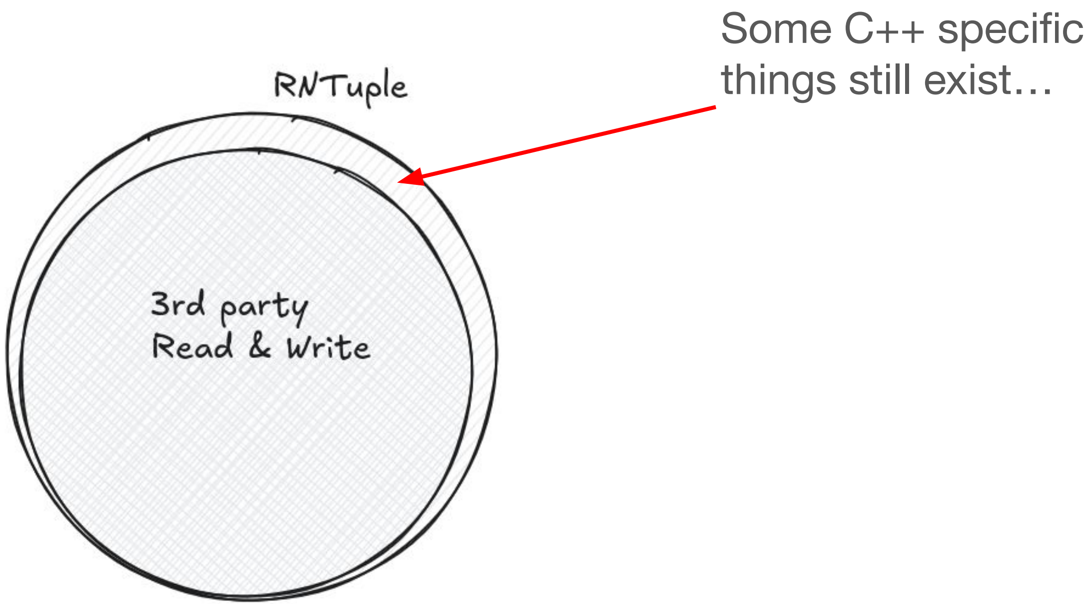
What is RNTuple

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



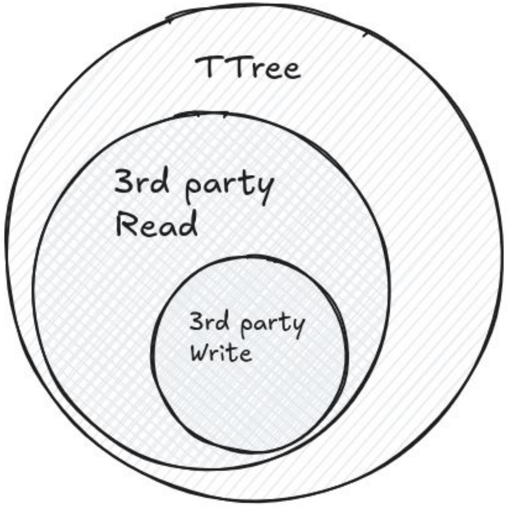
What is RNTuple

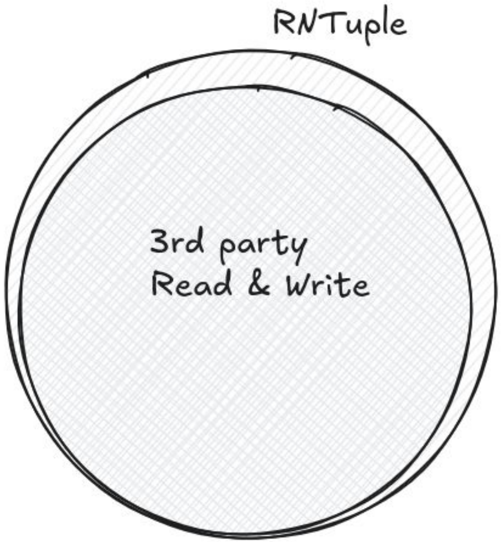
- In RNTuple, we can expect much more uniform compatibility thanks to specification-oriented design:

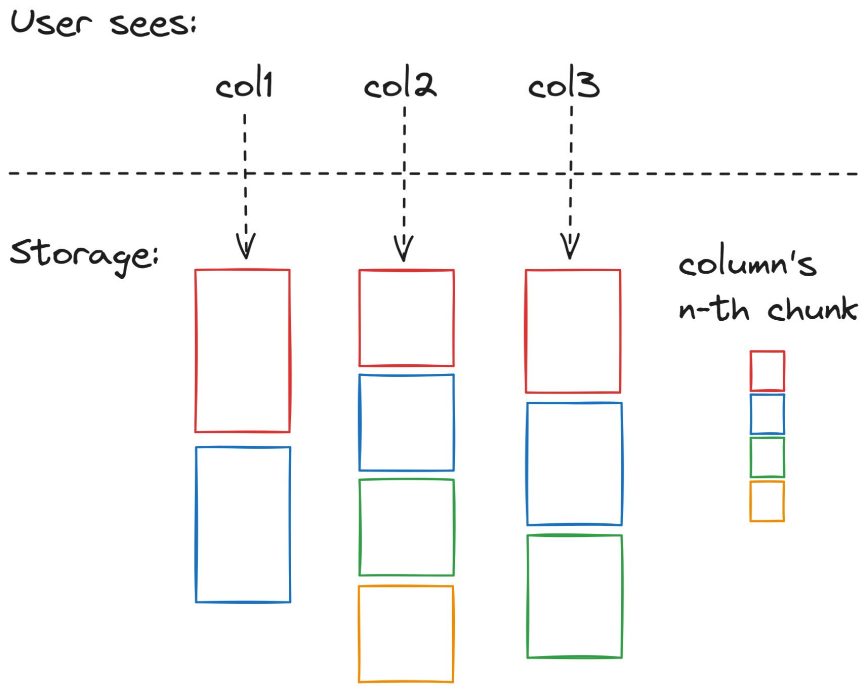


What is RNTuple

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



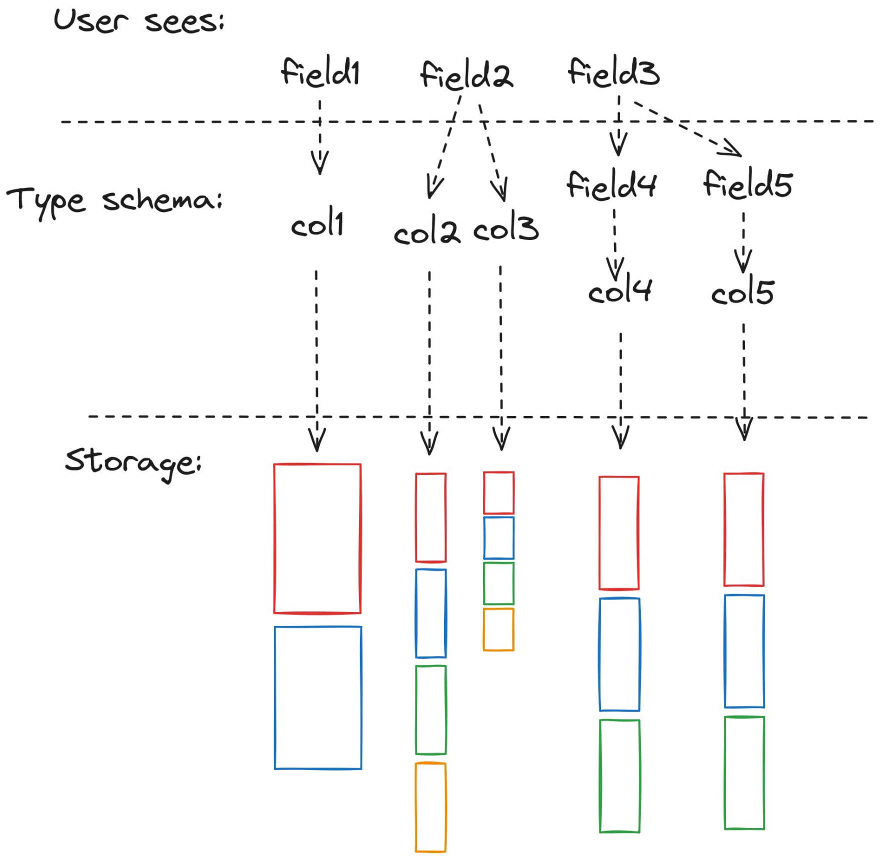




In **TTree**, every column the user sees correspond to one group of storage units.

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

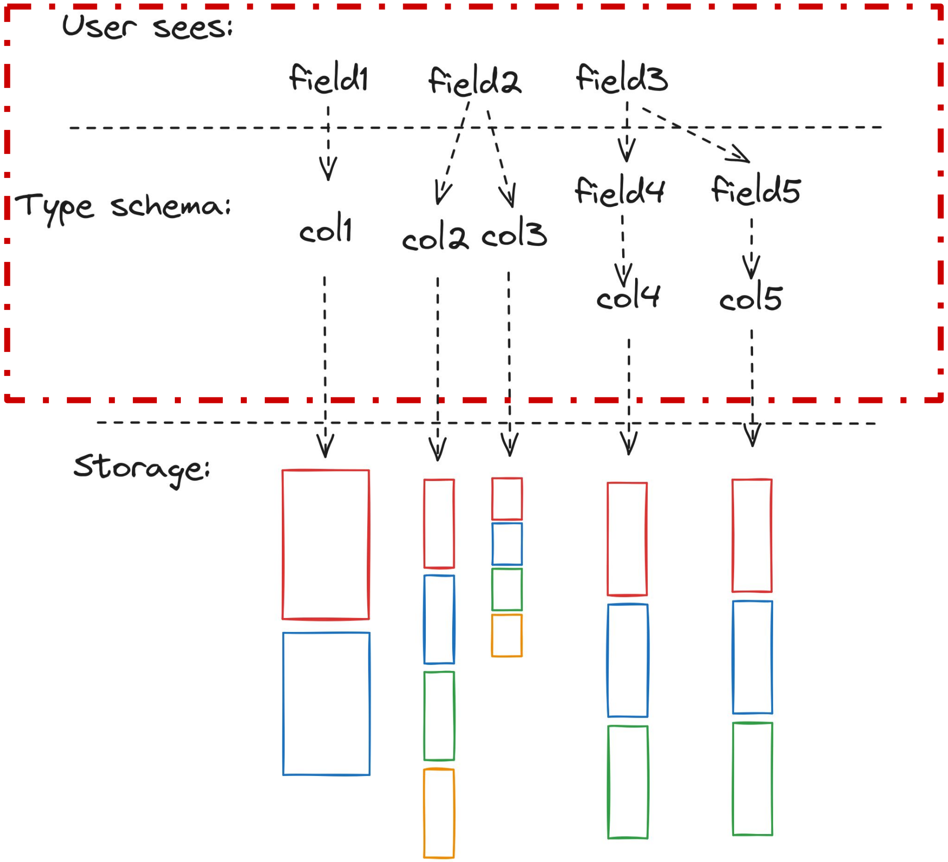
❖ **RNTuple's** design is more similar to Apache Parquet/Arrow(Feather):



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

This allows better compression efficiency and uniform schema composition rule.

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



The most challenging part is how to parse (for reading) or construct (for writing) the type schema.

- What's special about RNTuple (short version) ✅
- Implementation highlights in UnROOT.jl
- Current status and exciting future

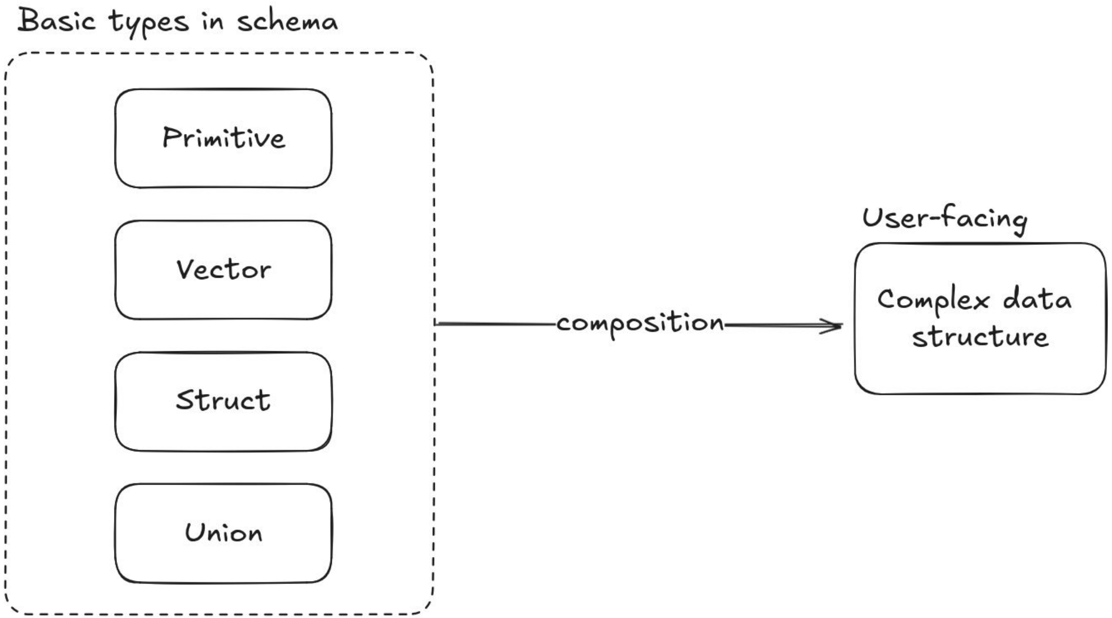
Implementation highlights

We highlight some Julia features that helped implememting read & write:

1. Multiple dispatch for implementing **type-space manipulations**
2. Type system for providing **flexible interface** downstream

1 - Type-space manipulation

- RNTuple types build up complex types via composition of a handful of basic types:



1 - Type-space manipulation

For both **read & write**, we implement for each “basic type” and let the dispatch system handle the composition.

- ❖ Read: assemble basic types to build complex type user sees
- ❖ Write: break down complex type into basic types of RNTuple

```
struct VectorField{O, T}
    offset_col::O
    content_col::T
end
isvoid(::Type{VectorField{N,T}}) where {N,T} = isvoid(T)

function _parse_field(field_id, field_records, column_records,
    offset_col = _search_col_type(field_id, column_records, al
```

```
# 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, alias_columns
    element_ids = findall(field_records) do field
        field.parent_field_id == field_id
```

2. Flexible interface via Julia type system

RNTuples are just tables, and each column, no matter how complex, can be seen as a vector. As an I/O package, we try not to get in the way of the users:

- Read: shouldn’t force special data structure onto users
- Write: shouldn’t require users to prepare their input into a narrow set of types

2. Flexible interface via Julia type system

Read: since each column is just an abstract vector, and the whole RNTuple is a table, user is free to use any container they want:

For-loop style

```julia
@threads for event in myTree
    hist = Hist1D(Float64; bins = 70:5:110)
    best_mass = Inf
    Z_m = 91.2 #GeV
    for i in idxs, j in (i+1):last(idxs)
        LV_i = lep_tlvs[i]
        PID_i = lep_pids[i]
```

Query style

```julia
using Query, DataFrames

@from event in myTree begin
    @let Njets = length(event.Jet_pt)
    @where Njets > 6
    @let Njets40 = sum(evt.Jet_pt .> 40)
    @select {Njets, Njets40, event.MET_pt}
    @collect DataFrame
end
```

User can write for-loop or use their favourite table-compatible ecosystem

2. Flexible interface via Julia type system

Write: anything table-like (with columns <:AbstractVector) can be ingested for free:

```
julia> x = [[1,2], [2,3,4]];

julia> x = [1:2, 2:4];

julia> x = VectorOfVectors([1:2, 2:4]);

julia> about(x)
2-element VectorOfVectors{Int64, Vector{Int64}, Vector{Int64}, Vector{Tuple{}}}
 Memory footprint: 24B directly (referencing 272B in total)
        data::Vector{Int64}    8B @ 0x000079a3d47c9a60 [1, 2, 2, 3, 4]
    elem_ptr::Vector{Int64}    8B @ 0x000079a3d47c9ac0 [1, 3, 6]
 kernel_size::Vector{Tuple{}} 8B @ 0x000079a4b6cbfa20 [(), ()]
```

2. Flexible interface via Julia type system

Write: after writing, they will all result in the same normalized column

```
julia> UnROOT.write_rntuple(open("./test.root", "w"), newtable;)

julia> LazyTree("./test.root", "myntuple")
 Row │ x
     │ Vector{Int64}
─────┼──────────────
 1   │ [1, 2]
 2   │ [2, 3]
```

- What's special about RNTuple (short version) ✅
- Implementation highlights in UnROOT.jl ✅
- Current status and an exciting future

Current status

⚠️ RNTuple v1.0.0.0 is yet to be released

	Read	Write
Primitive types	✅	✅
Vector	✅	✅
Struct	✅	🚧
Union	✅	🚧

What does this mean concretely?

Current status

Read: you can read basically* anything. (except byte blobs/legacy ROOT streamer)

```
├─ 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") ⇒
```

Example from ATLAS PHYSLITE *format*

Current status

Write: covers end-user analysis (private ntuple) usages such as CMS nanoAOD.

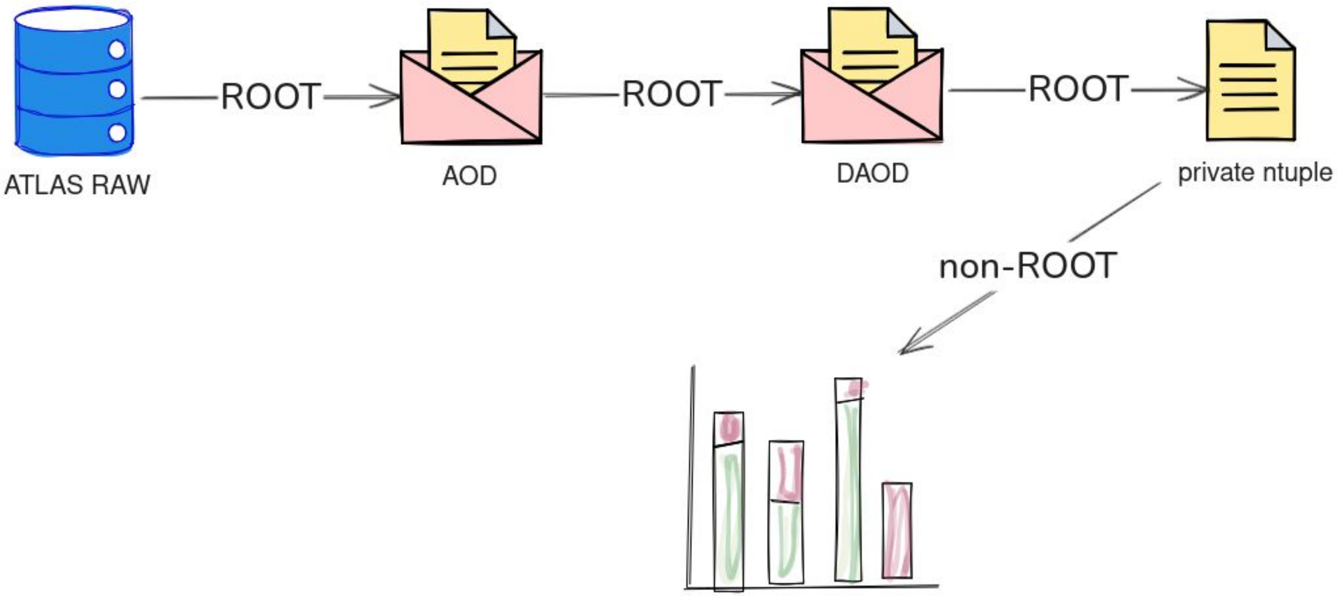
Concretely: all numerical primitive types, and Bool, String etc. As well as Vector of any of the primitive type (and doubly vector too etc.)

```
julia> t1 = LazyTree("./test/samples/NanoAODv5_sample.root", "Events");

julia> UnROOT.write_rntuple(open("./nanoAOD_rnt.root", "w"), t1;)

julia> t2 = LazyTree("./nanoAOD_rnt.root", "myntuple");

julia> isequal(DataFrame(t1), DataFrame(t2))
true
```

Converting NanoAOD from TTree to RNTuple in Julia; API subject to change

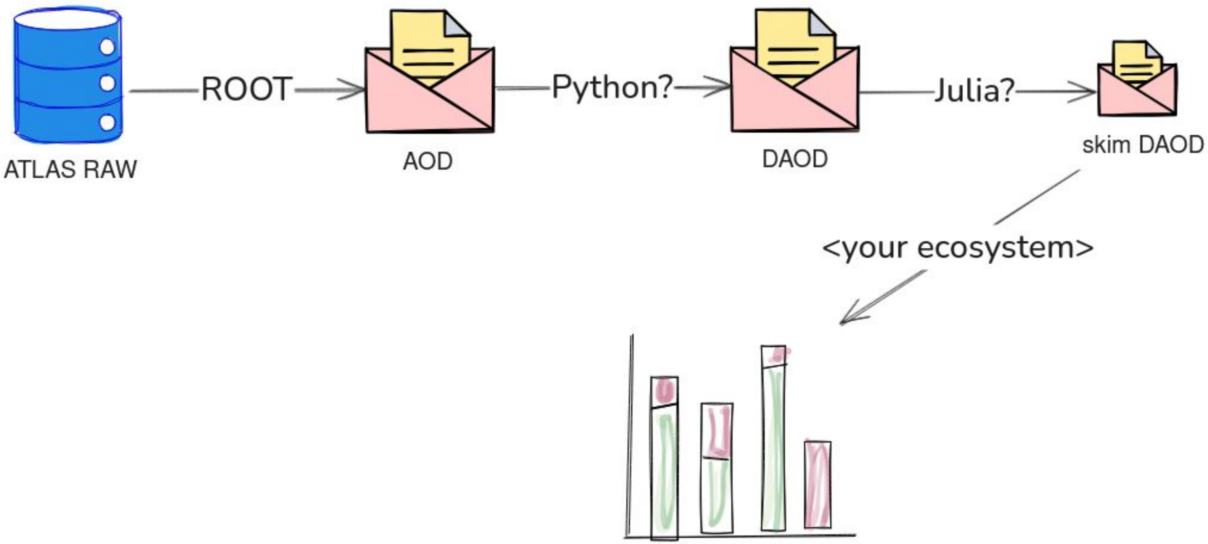
Exciting future

- ❖ Before RNTuple, UnROOT.jl has been successfully used for end-user analysis
  - ➢ Soon first published ATLAS paper



Exciting future

- ❖ Before RNTuple, UnROOT.jl has been successfully used for end-user analysis
  - ➢ Soon first published ATLAS paper
- ❖ With RNTuple, one can seamlessly implement many data pipeline steps in Python/Julia



Summary

- Pre-RNTuple, UnROOT.jl has only been useful for end-user analysis
- With RNTuple, much greater universal data compatibility between libraries
- Ready for experimental integration in larger data pipelines when stable RNTuple releases.

Backup

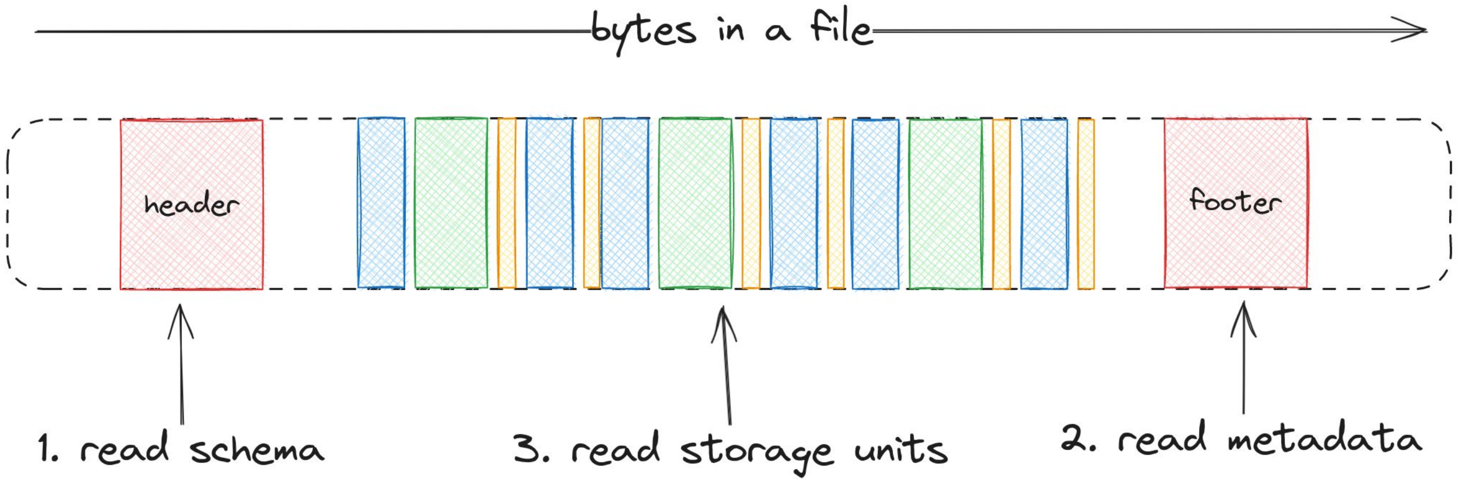
RNTuple is still evolving:

- 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.

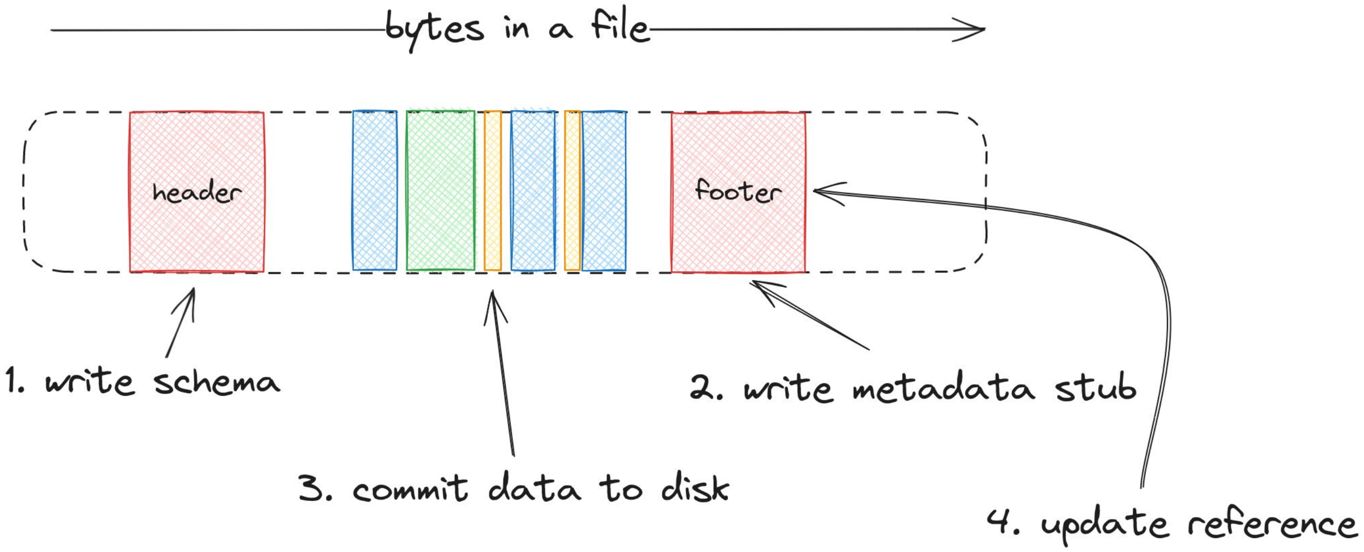
RNTuple writing strategy:

- Writing is very different from reading, in fact, almost no code can be reused.
- Information flow **during reading**:



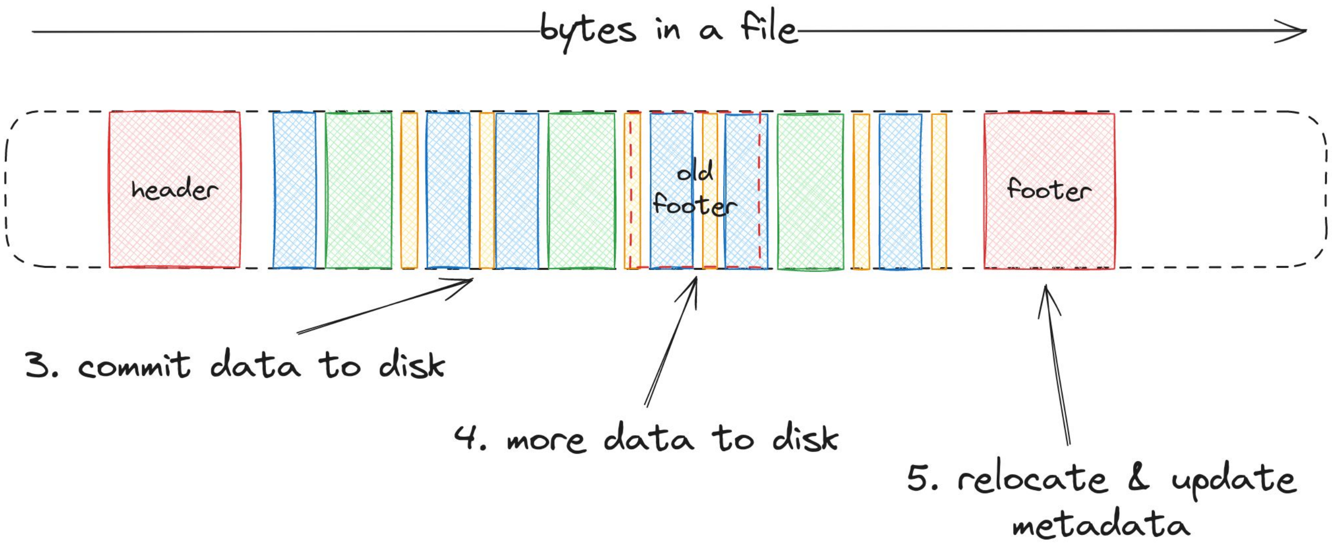
RNTuple writing strategy:

❖ **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:



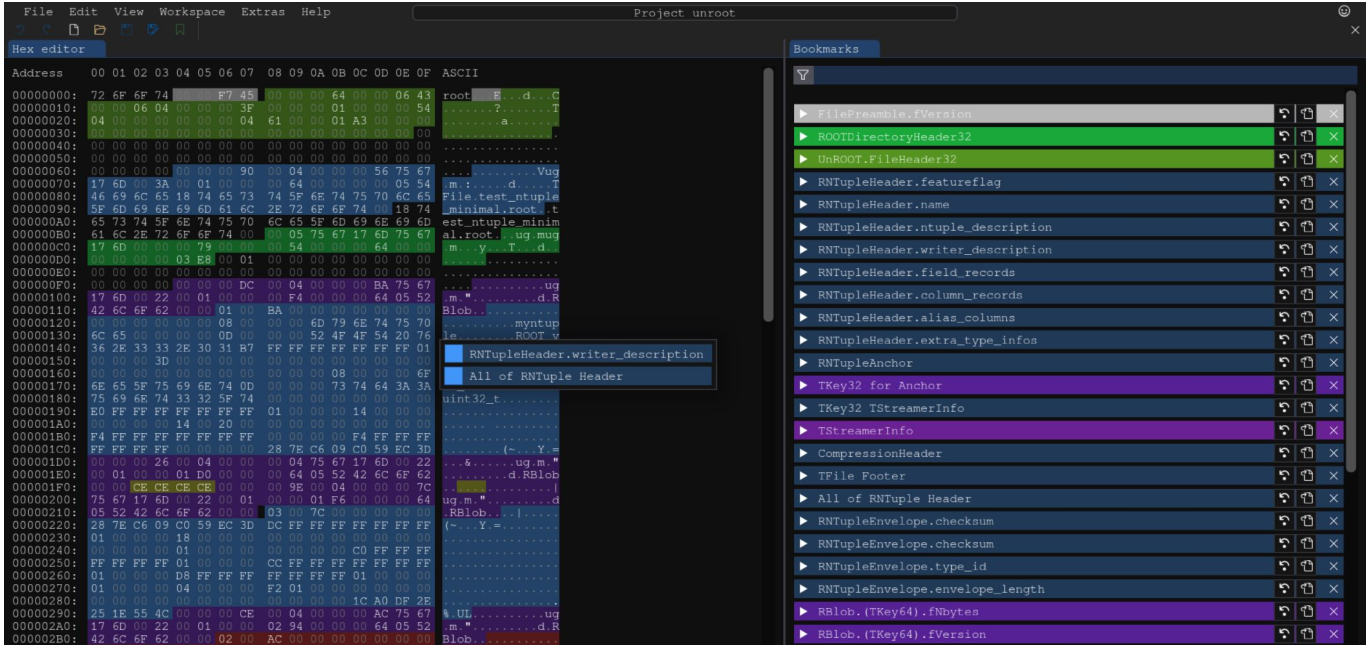
Development plan:

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 etc. (#349, #356) ✅
3. “Advanced” features: All 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:

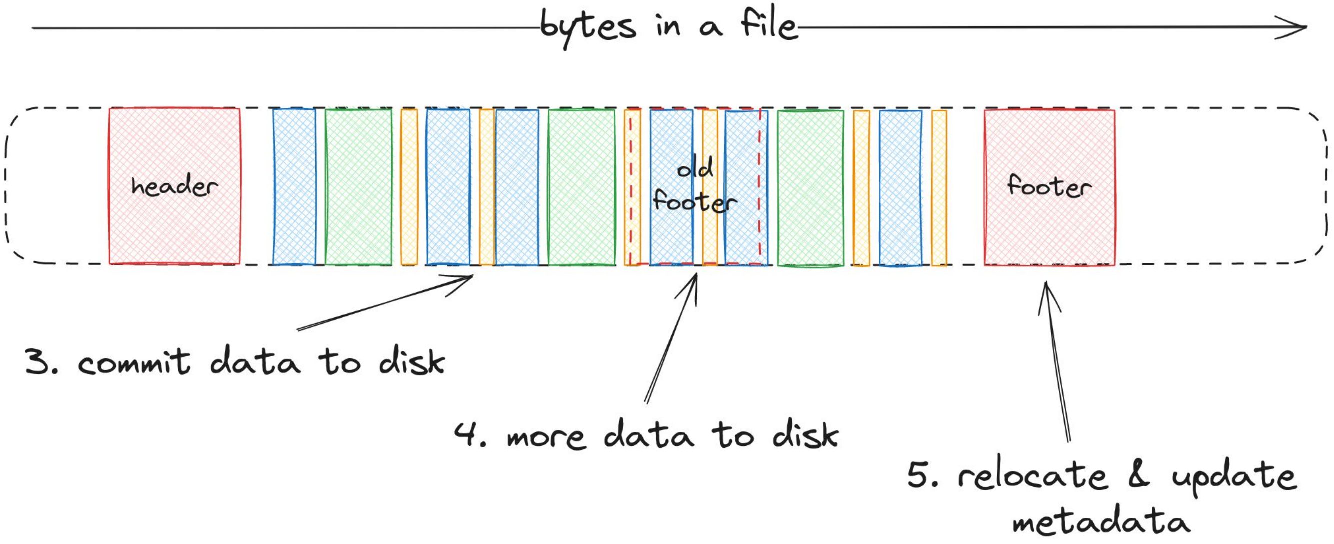


RNTuple writing: #1

- ❖ 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.



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:

❖ 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
```

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:

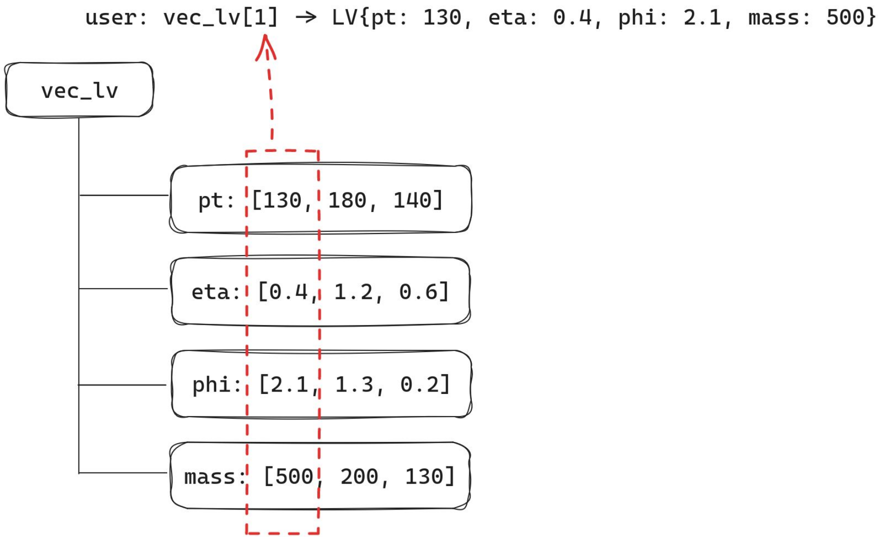
```
content = [12, 14, 17, 19, 21]
offset  = [0, 2, 2, 5]
```

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

"Content and offset" for jagged vector, similar to ArraysOfArrays.jl

RNTuple reading: type schema

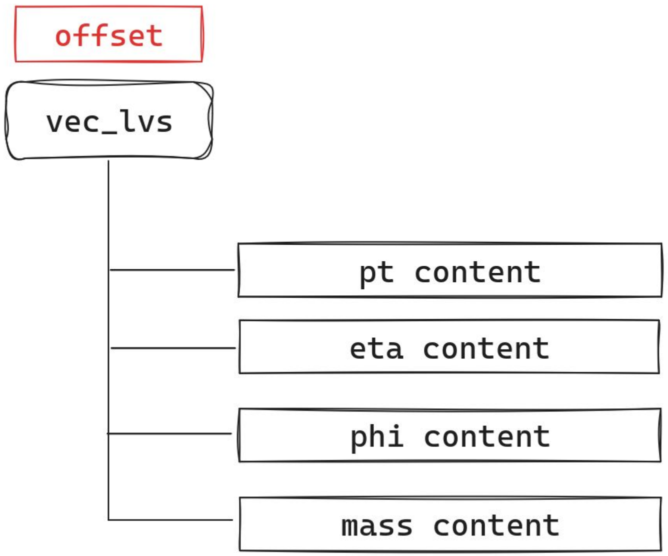
- The "struct" by itself is encoded using "struct of arrays" approach:



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"