Data analysis in Julia with RNTuples and plan for a Julia writer

Jerry Ling (Harvard University)

Dec., 2023

Learning Material

- Jakob Blomer's slides at JuliaHEP workshop in Sep 2021
- RNTuple specification
- write-up in PR Fully support RNTuple reading #200

Note: The pending RC2 change will be breaking but the broad-stroke concepts remain.

Overview

1. Introduction
2. Benefits for users (and developers)
3. Technical walk-through of RNTuple
4. Next step: writer

Intro: Why TTree → RNTuple?

TTree has been serving the community for a long time (27 years). Why change now? It seems to boils down to a few reasons:

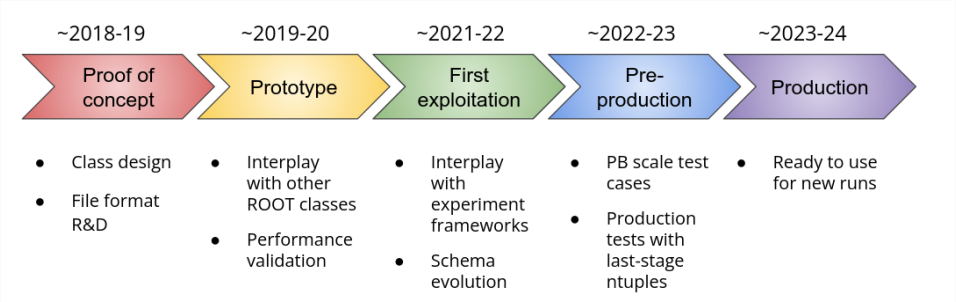
1. TTree had a LOT of special cases (e.g. singly, doubly, and triply jagged branches didn't use the same pattern) and it's getting harder to maintain over the years to add new support
2. These implementation hacks lead to inefficiency when storing and reading (nested) data collection (e.g. TObject serialization waste of metadata)
3. Out-dated designs: big-endian, hard to control I/O memory due to lack of "cluster or row group" support (exists but not enforced).

In conclusion: RNTuple will bring faster and better data type support for all HEP use cases.

Intro: Timeline of RNTuple

The ROOT Team views the RNTuple as a Run4 technology. Thus, "now" is a pretty good time to do alternative implementation (i.e. in a different language).

- as a crosscheck for the RNTuple spec: does the actual implementation adhere to the spec?
- give time for both ROOT and alternative implementation to mature before mass adoption



Intro: Comparing TTree and RNTuple

Using nanoAOD (ntuple-like, used by CMS, only flat and singly jagged branches) as performance benchmark,

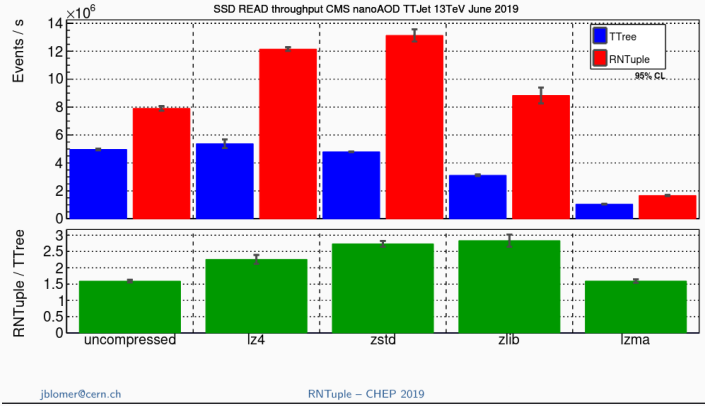


Figure 2: Reading performance comparison under different compression algorithms

User perspective: Comparing TTree and RNTuple

An RNTuple will be able to (recursively) store more (weird) C++ STL containers, e.g.:

- `std::pair<T1, T2>`
- `std::tuple<T1, T2, …, Tn>`
- `std::variant<T1, T2, …, Tn>`
  - Also known as `Union`
- For example you can have:

  `std::vector<std::variant<std::string, int32_t, float>> [1.0, "hi", "but why", 42]`

User-defined class must have fields that are RNTuple I/O compatible (finally, everything has to look like data-`struct` to be compatible, limitation in a healthy way)

User perspective: A Future of Full Interpolation

Problems:[a]

1. a lot of implementation-depenent stuff in reading and writing, hard to layout what exactly we support
2. bad user experience (e.g. “oh, didn’t know I can’t write back to .root I need to pass output to someone”)
3. high maintenance because 1.

[a] ref

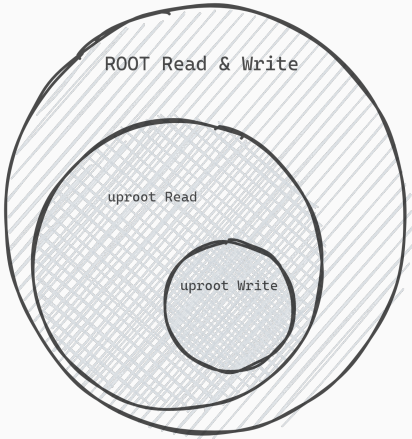


Figure 3: TTree support

What I/O for RNTuple looks like (as of 2023)

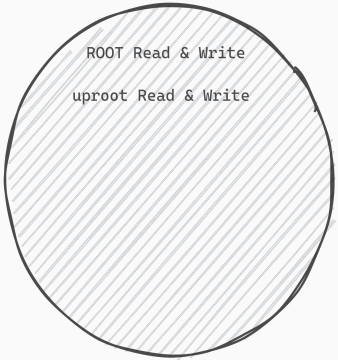


Figure 4: RNTuple best senario

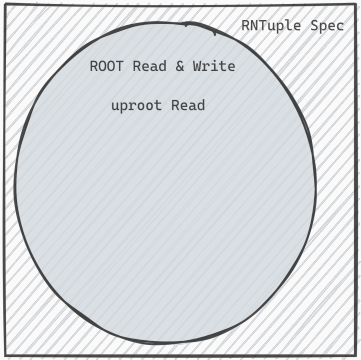


Figure 5: RNTuple reality, Julia same as uproot

Only weeks of development allows us to fully read almost everything! Unthinkable for TTree.

From Julia side: it just works

- Every column (top-level `field`) is still `<:AbstractVector`
- Code works for TTree -> works for RNTuple, in UnROOT.jl

From Julia side: Query vs. Loop

While for-loop always works,

```
for evt in table
    #...
end
```

Query-like APIs can make large analysis more structured, example benefits:

- tie-in systematics name handling (propagate to histograms)
- more amenable to compute-graph manipulations

We have `Query.jl` that can lump all queries into a single loop for simple cases. Open question: what's our priority in user analysis space?

Walk-through Step 0: Overview of .root file(system)

As you may know, `.root` is closer to a file system than a file.

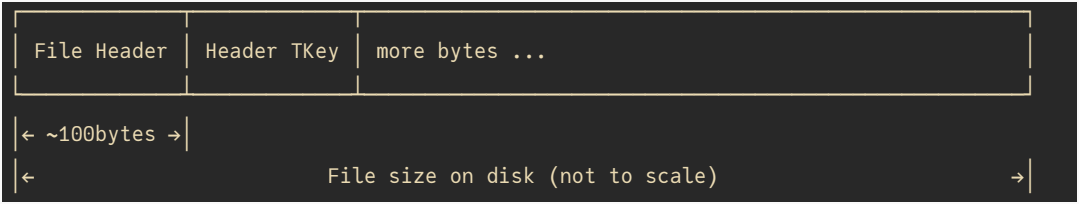
- All the bytes are self-descriptive
- Can store different types of data (TH1X, TTree, TObjString, RNTuple, Image)
- Can nest TDirectory within TDirectory
- Can look up objects by name without reading everything
- Reading data objects means to chase pointers to bytes (TKey)

However, RNTuple is largely independent of this ROOT legacy, it doesn't use any of the classical ROOT stuff (e.g. TStreamer, TDirectory) once we are "inside" an RNTuple.

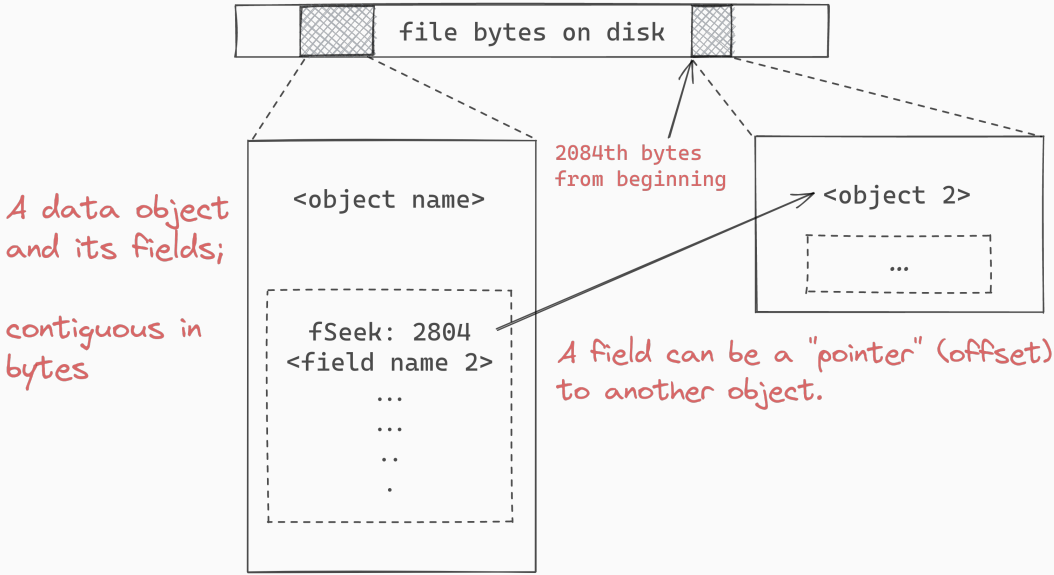
To explain how RNTuple works we will go through steps involved in reading an RNTuple.

Step 1: Finding RNTuple inside a .root file

Because `.root` is a file system and the RNTuple lives in it, we still need a little legacy ROOT logic to find it. Start with the entire `.root` file on disk:



Notations


file bytes on disk
2084th bytes
from beginning
A data object
and its fields;
contiguous in
bytes
<object name>
fSeek: 2804
<field name 2>
...
...
..
.
<object 2>
...
A field can be a "pointer" (offset)
to another object.

Step 1: Finding RNTuple inside a .root file

Once we have the header TKey, the rest can be summarized as the following:

This leads us to the gate object: RNTuple anchor
Once pass the gate (the anchor), everything will be in the "new" logic: little-endian, no more TKey, TStreamer, TDirectory look up. And we're reading to parse RNTuple from scratch.
While many of them are also possible to read/write with TTree (except `std::variant`), they are done mostly on a case-by-case basis and un-obvious how the

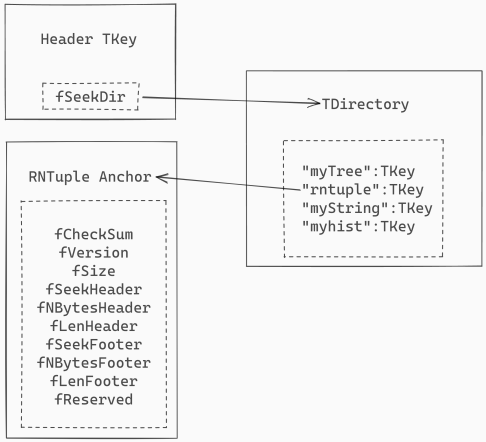


Figure 6: From .root to RNTuple anchor

Step 2: Parse Header + Footer

The anchor only leads us to the RNTuple header and footer (they contain metadata like schema), let's use header as an illustration.

Three fields in the anchor are related to header:

- `fSeekHeader` – the offset to the first byte of the header
- `fNBytesHeader` – the number of bytes of the header chunk in file
- `fLenHeader` – the number of bytes of the header **after decompression**

this implies that if `fNBytesHeader == fLenHeader`, we have uncompressed header in this file, this comparison is a common pattern in streaming I/O.

Rinse and repeat, and you get the entire schema of the RNTuple.

Step 3: Understand the Schema of RNTuple

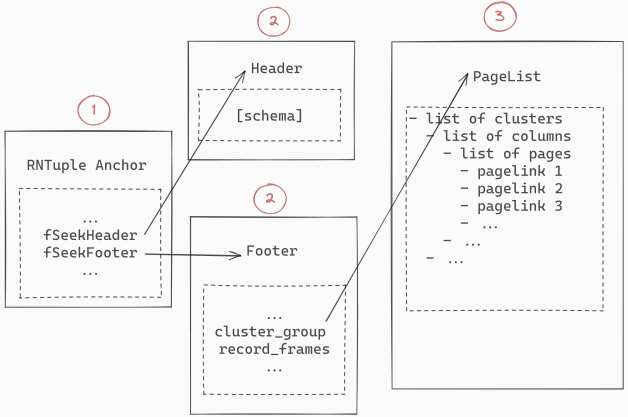


Figure 7: anchor to header/footer/page list

1. Find anchor in `.root` file
2. Parse header and footer meta data (schema)
3. Materialize `PageList` object and read actual data from pages (think basket)

The main point we're about to demonstrate is the RNTuple schema is compatible with Awkward Array. We will use an example data set and work through a few examples.

Step 3: The fields and columns in RNTuple Schema

Imagine your have a RNTuple that is:

Trigger	MET	lep_Pids
#Bool	*#Struct*	*#Vector{Int}*
true	(E = 530.3, ϕ = 2.3)	[11, 13, -13]
true	(E = 752.1, ϕ = -0.7)	[11, -11]
false	(E = 170.9, ϕ = 1.2)	[11, -11, -11, 11]

You might want to say it has three *columns*, but it actually has 6 fields and 5 columns. The `Trigger, MET, lep_Pids` are 3 (top) fields, not columns. In other words, users will always address each top field by name.

Step 3: Fields and Columns Records

Both the field and column records are stored in the header we just parsed. They look something like this when viewed directly.

Field records:

```
> rn.header.field_records
6-element Vector{FieldRecord}:
 parent=00, role=0, name=Trigger , type=bool
 parent=01, role=2, name=MET     , type=MET
 parent=02, role=1, name=lep_Pids, type=std::vector<std::int32_t>
 parent=01, role=0, name=E       , type=float
 parent=01, role=0, name=ϕ       , type=float
 parent=02, role=0, name=_0      , type=std::int32_t
```

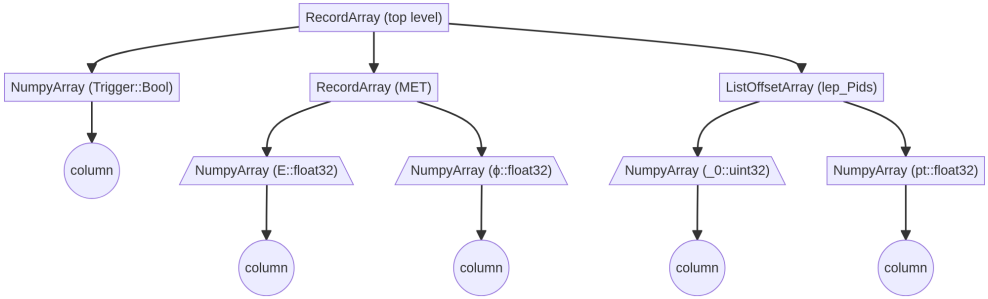
Step 3: Fields and Columns Records

Column records:

```
> rn.header.column_records
5-element Vector{ColumnRecord}:
 type=06, nbits=01, field_id=00, flags=0
 type=02, nbits=32, field_id=02, flags=5
 type=08, nbits=32, field_id=03, flags=0
 type=08, nbits=32, field_id=04, flags=0
 type=11, nbits=32, field_id=05, flags=0
```

Step 3: Schema as Awkward Form

If you're familiar with Awkward, here's a mapping from the RNTuple to an Awkward array, it happens so that you can map any RNTuple schema into Awkward, and it's the implementation strategy for Uproot:



Step 3: Schema as tree

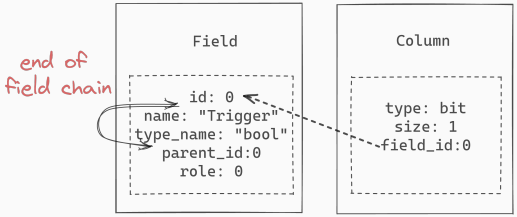
```
> rn.schema
RNTupleSchema with 3 top fields
├─ :Trigger ⇒ Leaf{Bool}(col=1)
├─ :MET ⇒ Struct
│         ├─ :E ⇒ Leaf{Float32}(col=3)
│         └─ :ϕ ⇒ Leaf{Float32}(col=4)
└─ :lep_Pids ⇒ Vector
               ├─ :offset ⇒ Leaf{Int32}(col=2)
               └─ :content ⇒ Leaf{Int32}(col=5)
```

As example, we will now manually parse fields to show how physical data is organized.

Step 3: Fields and Columns

The `Trigger` field is the most simple example, just a flat field:

```
# Field record implicit id = 0
parent=00, role=0, name=Trigger , type=bool
# Column record
type=06, nbits=01, field_id=00, flags=0
```



Step 3: Fields and Columns by example

For a simple struct field (`MET`), it needs N columns, N is the number of data fields of the struct:

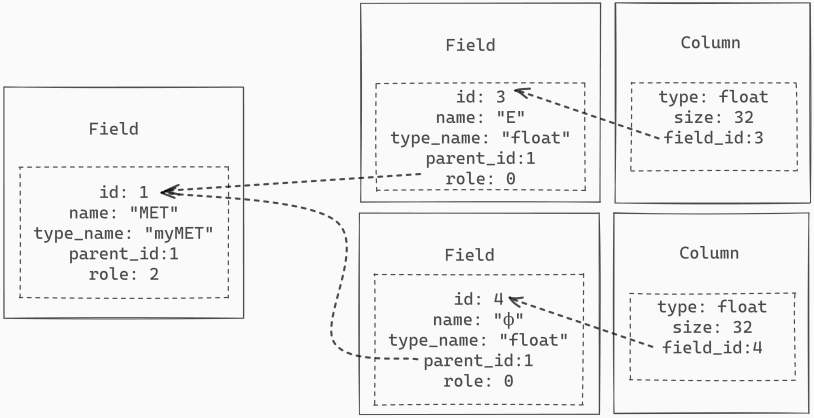


Figure 8: Field and column associated with MET

Step 3: Fields and Columns

For a singly jagged field (`lep_Pids`), it needs two columns, to represent `[[11,11], [], [13]]`, you have: 1) offsets: `[0, 2, 2, 3]`; 2) content: `[11, 11, 13]`.

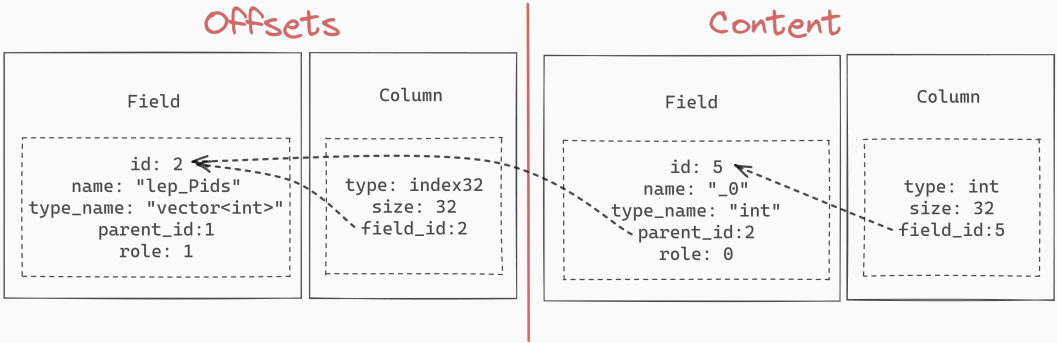


Figure 9: Field and column associated with lep_Pids

Step 4: Putting it all together

1. The RNTuple header tells you on how *fields* and *columns* should be interpretated.
2. The footer tells you where to find *pageslist* (somewhere else in the file).
3. The *pagelist* tells you where to find *pages* (which have actual data) for each column.

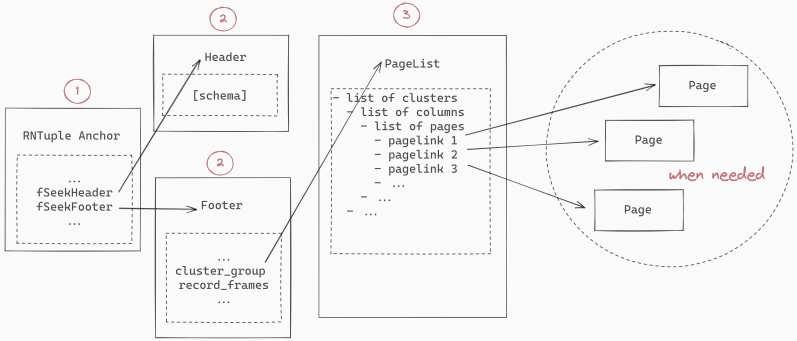


Figure 10: Finally

Step 4: Putting it all together

To show what the `PageList` ("list of list of list") is for:

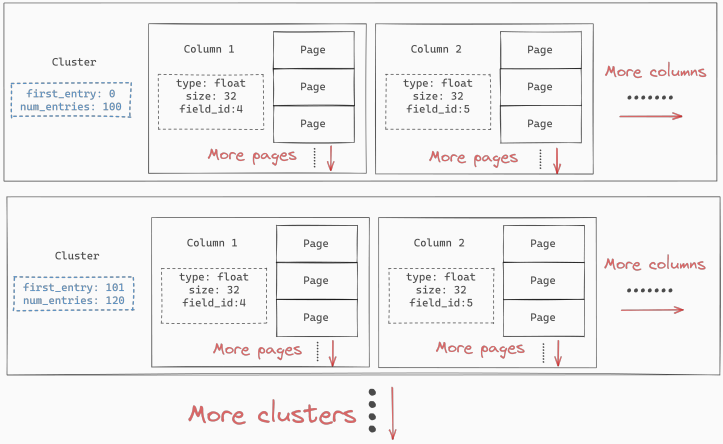


Figure 11: Triply-nested list

Step 5: Handle Cluster Group

Each `PageList` is responsible for a "Cluster Group" (which has multiple clusters, corresponding to the outer most list in the triply nested list).

In the rare case of having more than one cluster group, use information from "Cluster Summary" to find the `n-th` cluster, and find the Cluster Group that contains it.

Comment: Implication on Reading Granularity

- In `TTree`, the granularity of reading is a `TBasket` from a `TBranch`. The basket payload has to be decompressed as a whole, and the payload will contain complete data for `N` events for that branch.
- Furthermore, the branch contains meta data on the "event range" of all its baskets to facilitate random reading.
- In `RNTuple`, the closest analogy to a "basket" is a "page", however, because "page" belongs to column, and a field visible to the user may have multiple columns, a given event may involve data across "page" boundary.
- In summary, for `RNTuple` the only way to be certain you have full data for `N` complete events is to read the entire cluster for the relevant fields(columns).

Comment: Connection to industry formats

It turns out RNTuple shares a lot of similar ideas to Parquet (in chunking/pagination) and Arrow (in schema tree):

RNTuple	Parquet	Arrow (in RAM)/Feather (disk)
field	column	field
column	–	array
cluster	row group	row group
page list	column chunk	record batch
page	page	buffer

Conclusion and next step

RNTuple as a format is a long-awaited evolution of TTree:

- Composable in types, allow succinct and more correct implementation
  - More "language-independent" if third-party developers attempt
- Better performance on real physics data with correct cluster size tune

From a user perspective:

- Reading of a large variety of types already functional in Python and Julia
  - The development is very efficient (less hours, more weird types)
- Future: a 100% compatibility in both reading and writing is possible

Near future: RNTuple Writer

First, I want to make the scope clear. Writing RNTuple means "able to write out a single RNTuple table embeded in a `.root` file".

Importantly, RNTuple has specification, the container `.root` file (i.e. `TFile`) does not. But ROOT team has signaled they're willing to freeze `TFile` specification to make claims like "compatible with RNTuple version X" sound.

Near future: RNTuple Writer

Reverse the "reading walk-through" actually requires completely different code paths:

- When reading a file, you read metadata first, and the content, recursively.
- When writing a file, you need to commit bytes to disk in reverse order.

This is because the metadata often "points" to content (either in type space or just disk offsets); you don't know what to write in metadata until pointee is frozen.

Challenges:

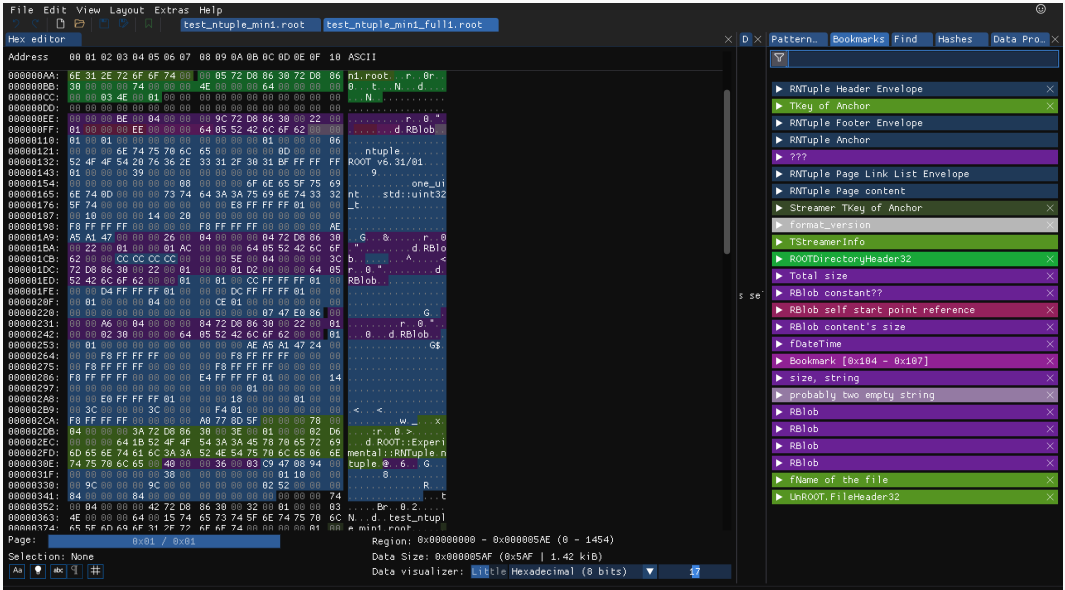
- you need to manage "empty slots" in your file byte blob
- when things don't fit, you need to shift bytes blobs around (most relevant if we allow appending)
- ROOT has hidden metadata, logically not useful for RNTuple, but lack of them would render the file illegal for ROOT to read.

Near future: Check List

- Individual components of RNTuple is understood (byte representation)
- Cascade writer for everything-RNTuple is easy (i.e. exists in my head)
- TFile-related less clear:
  - How does TStreamer work?
  - RBlobs rule?

I have looked at hex dump of simple ROOTFiles: Some possible temporary workarounds: copy bytes blocks from legal ROOT files.

HexDump


File Edit View Layout Extras Help
test_ntuple_min1.root
test_ntuple_min1_full1.root
Hex editor
Address 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 ASCII
Pattern..
Bookmarks
Find
Hashes
Data Pro..
RNTuple Header Envelope
TKey of Anchor
RNTuple Footer Envelope
RNTuple Anchor
???
RNTuple Page Link List Envelope
RNTuple Page content
Streamer TKey of Anchor
format_version
TStreamerInfo
ROOTDirectoryHeader32
Total size
RBlob constant??
RBlob self start point reference
RBlob content's size
fDateTime
Bookmark [0x104 - 0x107]
size, string
probably two empty string
RBlob
RBlob
RBlob
RBlob
fName of the file
UnROOT.FileHeader32
Page: 0x01 / 0x01
Selection: None
Region: 0x00000000 - 0x00005AE (0 - 1454)
Data Size: 0x00005AF (0x5AF | 1.42 kiB)
Data visualizer: Little Hexadecimal (8 bits)