Reading RNTuple from uproot

Jerry Ling (Harvard University)

Oct. 6, 2022

Bread Crumbs

- ROOT Team's talk slides at CHEP 2019
- Jakob Blomer's slides at JuliaHEP workshop in Sep 2021
- ROOT's official specification on RNTuple
- My PRs to uproot as IrisHEP fellow with (maybe) helpful long comments:
  - Primitive Support for RNTuple #630
  - Implement stl containers for RNTuple #662
  - Multiple clusters support for RNTuple #682
  - feat: Support writing of RNTuple #705

Why TTree → RNTuple?

TTree has been serving the community for a long time (27 years). And we know many reasons by ROOT team (in particular, performance).

But from an outside (of ROOT) prespective, RNTuple is an oppurtunity to completely harmonize I/O landscape across languages.

What I/O for TTree looks like

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

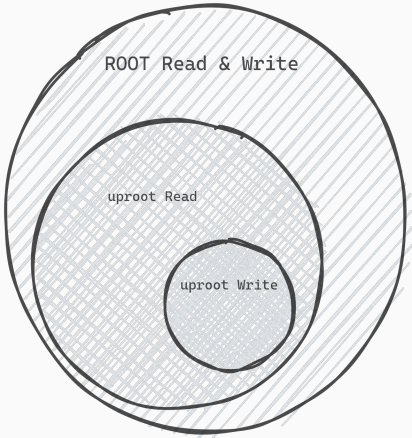


Figure 1: TTree support

What's different about RNTuple

RNTuple has a spec. It (could) enable us to:

1. implement reader and writer systematically once and for all
2. or at least maintenance would be much more systematic: "just fix bugs that doesn't agree with spec"
3. more confident in unit tests see later section
4. clear for all users across language what can be stored and how it will be stored (e.g. no more `float [N]` vs. `std::vector<float>`)

What I/O for RNTuple (could) look like

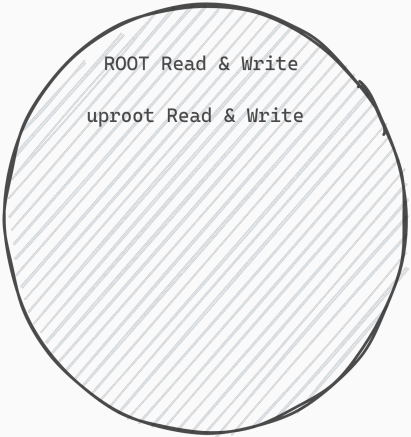


Figure 2: RNTuple best senario

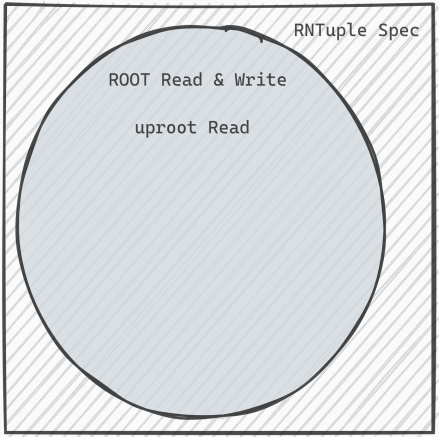


Figure 3: RNTuple reality

What I/O for RNTuple in Oct 2022

- Most things are implemented as advertised! Major shout out to ROOT I/O team.
- A few items from the spec are not implemented (e.g. sructure role `0x04`, The field is a reference (pointer)).
- 100% readable from uproot is not mathematically proven just via some limited unit tests

Examples of types uproot can read

from uproot RNTuple unit test:

```
string
vector<int>
vector<vector<int>>
vector<string>
vector<vector<string>>
variant<int32, string>
vector<variant<int, string>>
tuple<int,string>
vector<tuple<int,string>>
```

More confidence in unit tests

For RNTuple unit tests, if:

- `T1, T2, T3, T4` works (out of 15 primitive types)
- and `vector<T2>` works
- and `tuple<T1, T2>` works

then I have really high confidence that these will work too:

- `vector<T3>`
- `vector<vector<T1>>`
- `vector<vector<vector<T1>>>`
- `tuple<T3, T4>`
- `vector<tuple<T3, T4>>`

Definitely can't say the same about TTree (i.e. 1-jagged and 2-jagged have special implementations).

Quick Recap of how RNTuple schema works

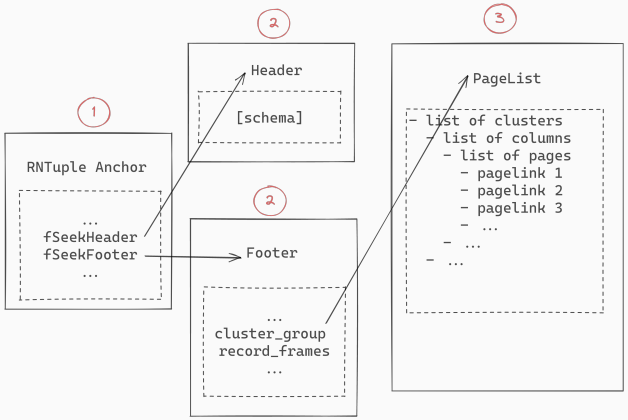


Figure 4: Anchor to header/footer/page list

While the word “column” makes people think of column in a spread sheet or a TBranch in a TTree, it is NOT what you think it is. To understand how the data is logically organized we need to know the relation between fields and columns.

The Schema of RNTuple: fields and columns

Imagine your have a RNTuple that is:

Trigger	MET	lep_Pids
true	(E = 530.3, ϕ = 2.3)	[11, 13, -13]
true	(E = 752.1, ϕ = -0.7)	[11, -11]

Let's look at the `MET` field as an example to see how RNTuple is actually a tree (data structure).

Fields and Columns by example

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

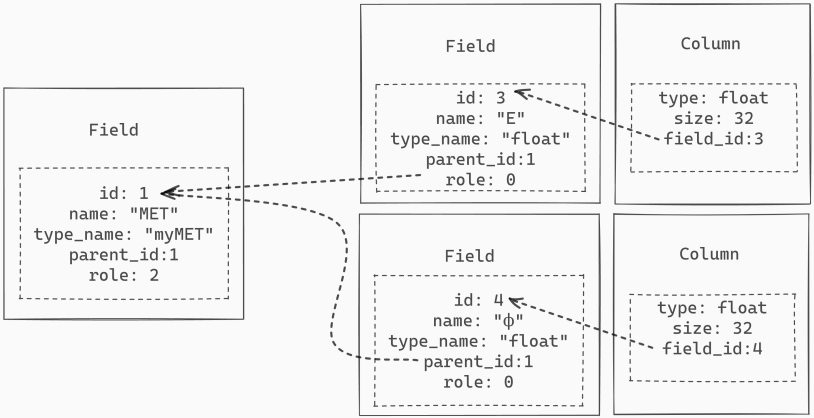


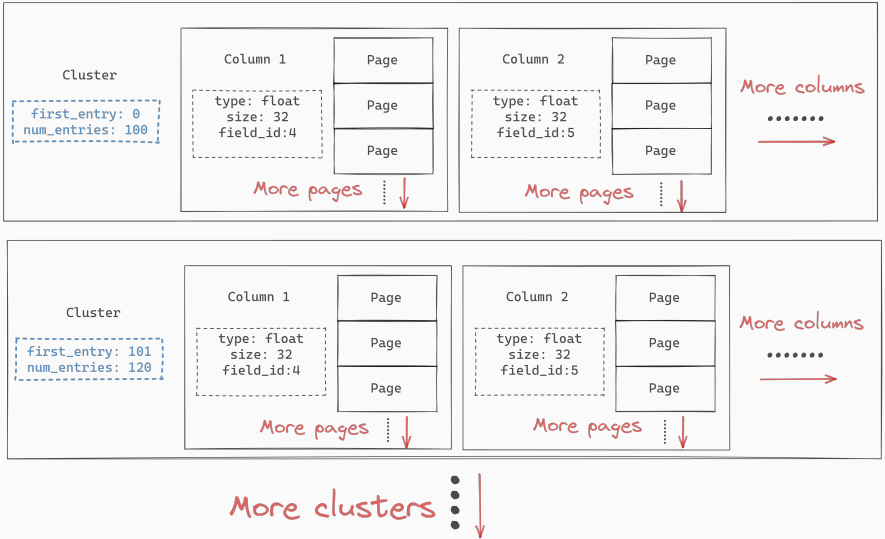
Figure 5: Field and column associated with MET

Connection to industry

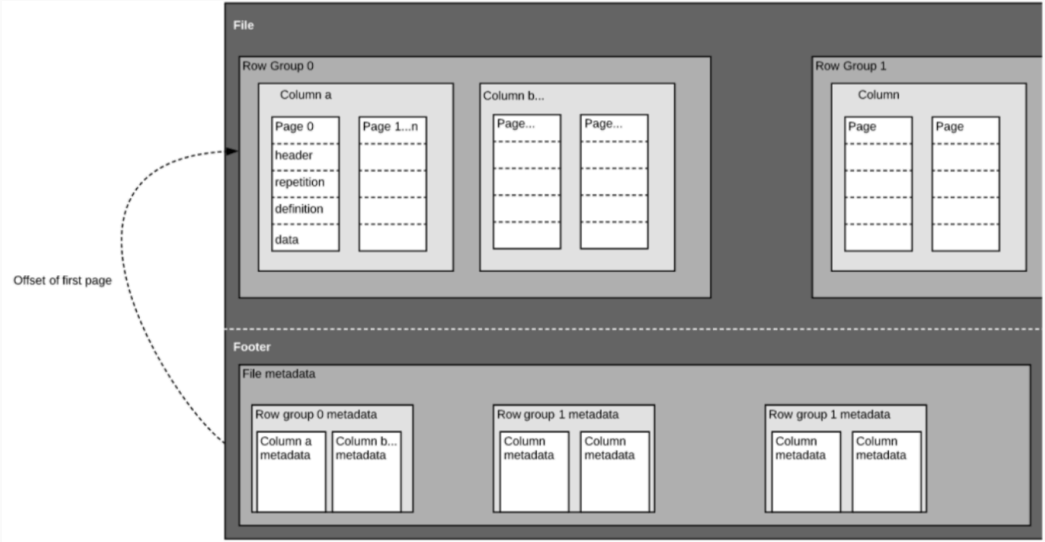
It turns out RNTuple shares a lot of similar ideas to Parquet (in chunking/pagination) and Arrow (in schema tree):

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

How similar? RNTuple



How similar? Parquet



Shema tree

But more importantly is how the schema works, in this regard, RNTuple is much closer to Arrow/Feather than Parquet, mainly because:

- Parquet doesn't have column and field, much like TBranch, it jams all nesting and metadata into the same column
- Arrow/Feather fields, which don't directly correspond to primitive data array, only exist for schema, similar to RNTuple (even primitive field has a column to it).
- Arrow/Feather has about the same kind of variation for primitive/logical types compared to RNTuple

But most importantly, since Awkward array adopts the same schema tree idea, we have a clear path towards how we should implement the reader.

Work needed for RNTuple Reader

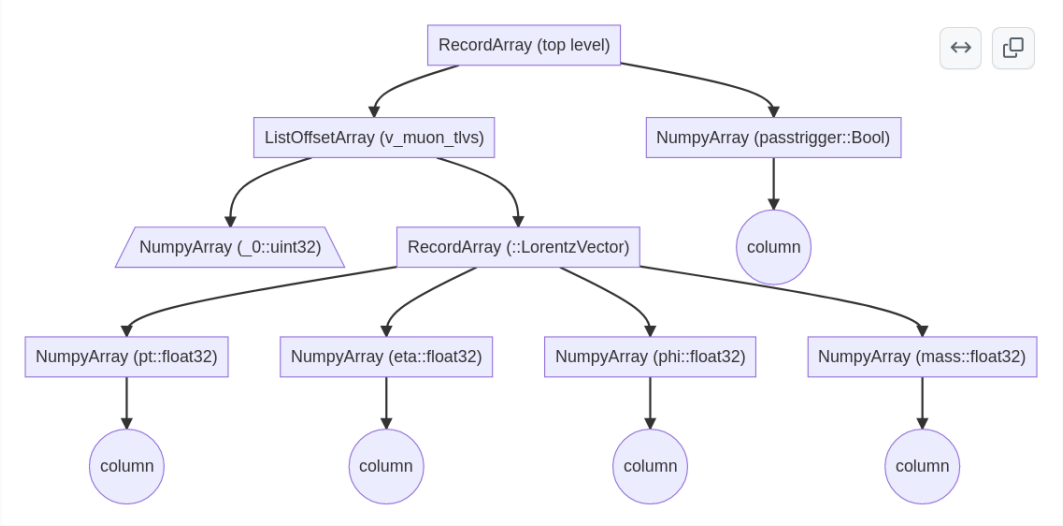
1. Parse Schema of RNTuple (header and footer content)
2. Build the schema tree of Awkward array by looping over fields and column descriptions
3. Whenever we run into logical types (struct, collection, union etc...), translate them into awkward array schema types (awkward `Form`)
4. Connect storage (columns) to the correct leaf node of Awkward schema (`NumpyArray`)
5. ????
6. profit

Examples of type translation

1. most prititive types have `numpy` dtype already (special case: Bit, Switch, split encoding)
2. mother of a collection $\longrightarrow$ `ak.ListOffsetForm`
3. mother of a record $\longrightarrow$ `ak.RecordForm`
4. mother of a variant $\longrightarrow$ `ak.UnionForm`

In retrospection, it was pretty easy, the harder part is to figure out what is what and debugging shcema parsing AND bytes reading at the same time unsure where the bug is.

Tree-like schema of RNTuple



Some observation on application

Entry range reading is pretty common (and is the only way to have reasonable performance in Python), but in this case the atomic unit to read is a whole cluster. This is because the `page`s (which contain primitive data) don't necessiarily end on entry boundary.

Another non-incremental difference from Apache Feather is the design of `Column Group` which allows different cluster to span different columns (where row group in Apache formats always span all fields/arrays). Theoretically, it would help when you have columns with drastically different "per entry" size (Bit vs. Float64). Remains to be tested if it saves time in the real world (i.e. if someone is reading trigger, they probably is also reading data).

Thanks!

Especially thanks ROOT I/O teams hard work and Jim for mentoring this project.

P.S Next up is a Julia reader before I forget all of this.