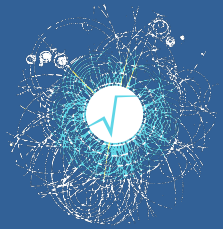
RNTuple: latest developments & the RNTupleProcessor



Florine Willemijn de Geus, CERN EP-SFT & University of Twente (NL)

for the RNTuple development team

EP R&D Software Working Group Meeting
January 15, 2025
florine.de.geus@cern.ch

EP R&D

UNIVERSITY OF TWENTE.

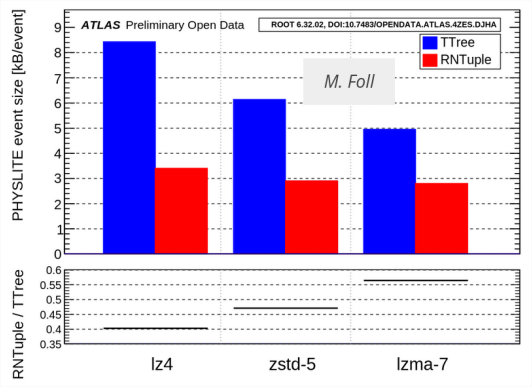
Introduction

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

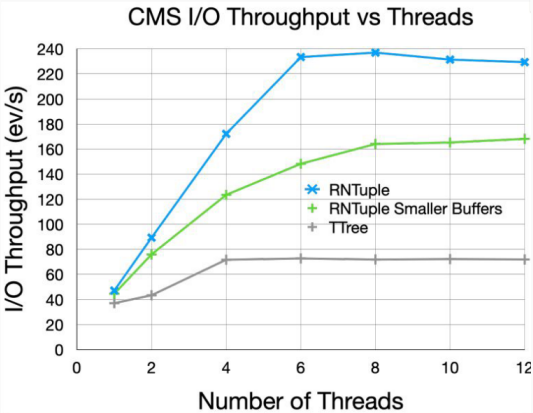
- Less disk and CPU usage through *smaller files* and *higher throughput*;
- Systematic use of *data checksums* and runtime *exceptions* to prevent silent I/O errors;
- Efficient support for *modern hardware*;
- Native support for local and remote *ROOT files* and *object stores*;
- Coverage of all of today's ***TTree*** *use cases* (integration in both Athena and CMSSW);
- A binary format defined in a dedicated specification.

The first production version of the **RNTuple** on-disk binary format is available in ROOT 6.34. The API will start moving out of `ROOT::Experimental` from ROOT 6.36 onward.

Recent performance results I

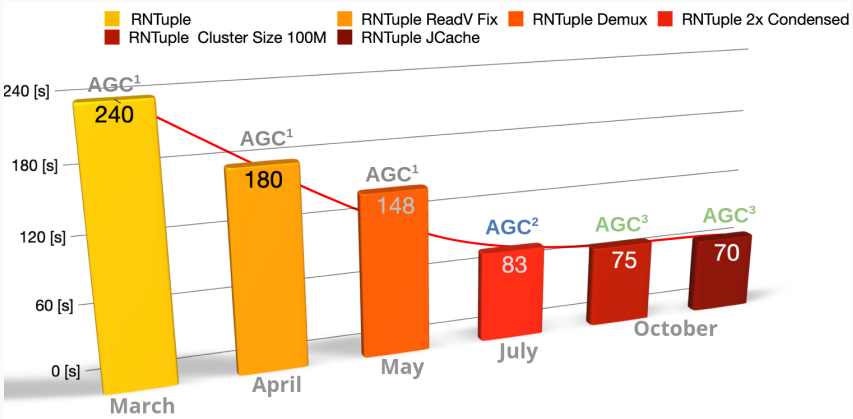


CHEP '24: *RNTuple: A CMS perspective*



CHEP '24: *ROOT RNTuple and EOS: The Next Generation of Event Data I/O*

Recent performance results II



CHEP '24: *ROOT RNTuple and EOS: The Next Generation of Event Data I/O*

Combining data sets with the **RNTupleProcessor**

Why do we want to combine data sets?

A data set is typically stored across multiple files (*samples*), but we want **seamless** event processing.

- ➔ need to be able to **vertically concatenate** samples.

Why do we want to combine data sets?

A data set is typically stored across multiple files (*samples*), but we want **seamless** event processing.

- ➔ need to be able to **vertically concatenate** samples.

1. Analysis may require objects not present in the compact data format;
2. Analyses could be sped up by storing and reusing (expensive) intermediate computation results.

- ➔ need to be able to **horizontally concatenate** samples.

Combining HEP data sets

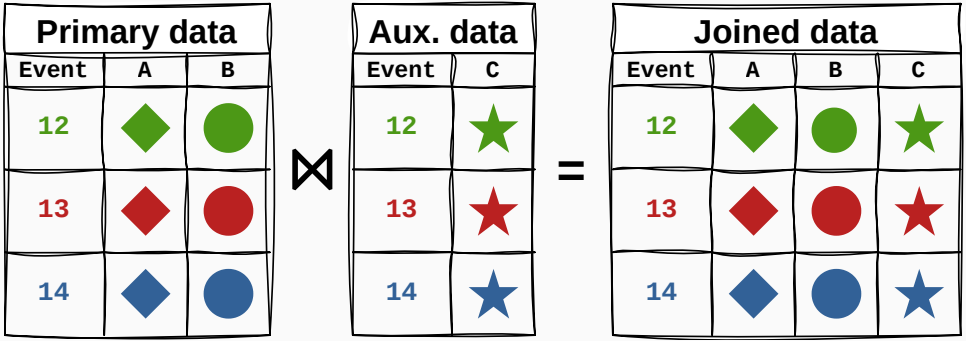
TTree has the ability to concatenate data sets in two directions:

1. *Vertically* through the `TChain` interface;
   - ➔ comparable to a SQL UNION ALL operation (*but not exactly*).
2. *Horizontally* through the `TTree::AddFriend` interface, possibly using a `TTreeIndex` for unaligned entries
   - ➔ comparable to a SQL JOIN operation (*but not exactly*).

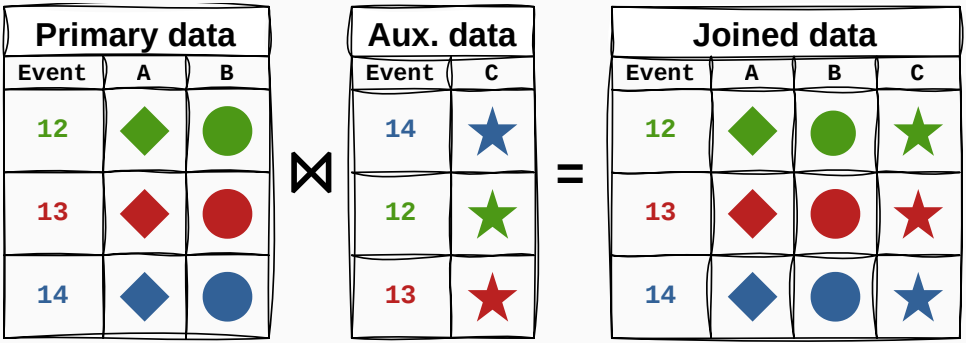
They can be combined using `TChain::AddFriend`.

Similar functionality is desired for **RNTuple**. We want to provide additional composition flexibility and above all, prevent users from accidentally getting erroneous data.

Data set joins: the ideal case

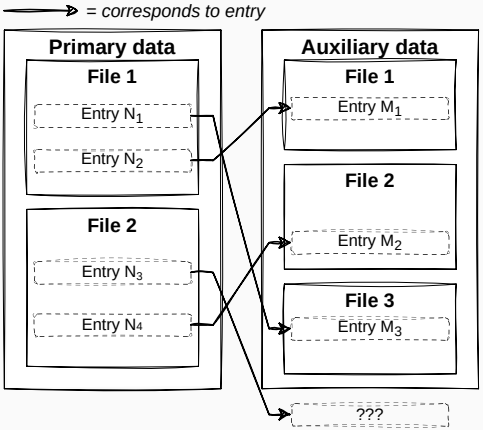


Data set joins: a realistic scenario



The caveats of unaligned data set joins

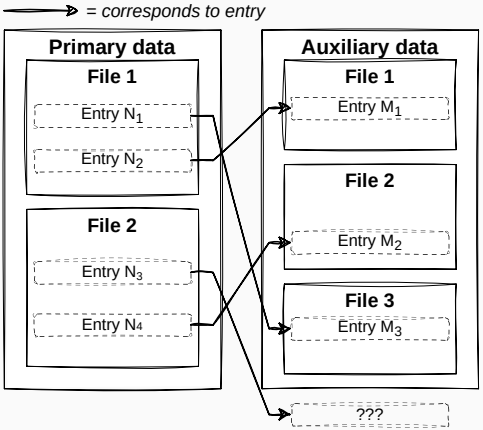
- Which events belong together?
  - Both false positives and negatives are unacceptable!
- What if the right-hand side event data is missing?
- What if my events are scattered across multiple files?
- What if want to distribute my analysis?



The caveats of unaligned data set joins

- Which events belong together?
  - Both false positives and negatives are unacceptable!
- What if the right-hand side event data is missing?
- What if my events are scattered across multiple files?
- What if want to distribute my analysis?

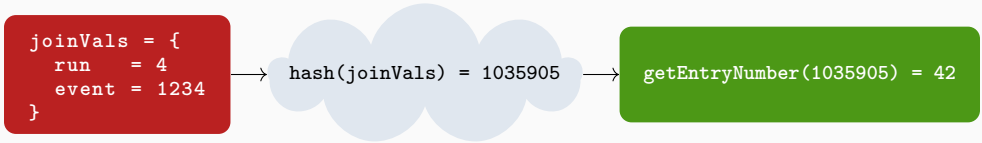
+ How to express all of this nicely?



Handling unaligned joins

When events between two data sets don't align on their entry numbers, we need a **join table**:

- Mapping between values of one or multiple *join columns* and corresponding entry numbers;
  - Support for up to 4 **integral-type** join columns;
  - Multiple column values are combined into a single hash.
- Built for the *auxiliary data set*;
- Probed using values from the *primary data set*.



The RNTupleProcessor

New data iteration model: **RNTupleProcessor**.

Responsible for handling chains and joins , in a unified way.

The RNTupleProcessor

New data iteration model: **RNTupleProcessor**.

Responsible for handling chains and joins , in a unified way.

```
std::vector<RNTupleSourceSpec> ntuples{
    {"myElectrons", "electrons1.root"}, {"myElectrons", "electrons2.root"}};
auto processor = RNTupleProcessor::CreateChain(ntuples);

for (const auto &entry : *processor) {
   std::cout << "pt = " << *entry.GetPtr<float>("pt") << std::endl;
}
```

→ See the ntpl012_processor_chain.C tutorial

The RNTupleProcessor

New data iteration model: **RNTupleProcessor**.

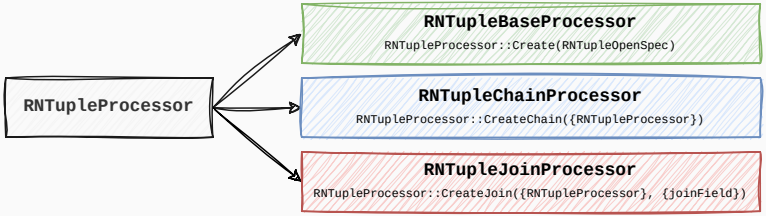
Responsible for handling chains and joins, in a unified way.

```
std::vector<RNTupleSourceSpec> ntuples{
    {"myElectrons", "electrons.root"}, {"myMuons", "muons.root"}};
auto processor = RNTupleProcessor::CreateJoin(ntuples, {"run", "event"});

for (const auto &entry : *processor) {
   std::cout << "electron pt = " << *entry.GetPtr<float>("pt") << std::endl;
   std::cout << "muon pt = " << *entry.GetPtr<float>("myMuons.pt") << std::endl;
}
```

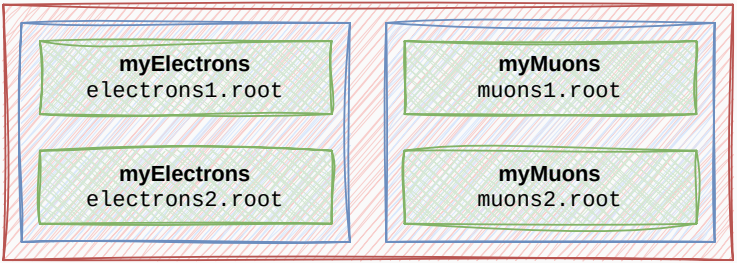
→ See the ntpl015_processor_join.C tutorial

Composability of the RNTupleProcessor



Each processor implements the same interface for loading entries, allowing for arbitrary composition ordering.

Chain-first approach

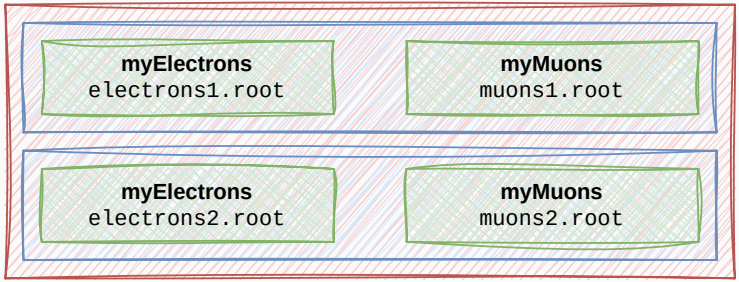



```
auto electrons = {Create({"myElectrons", "electrons1.root"}), Create({"myElectrons", "electrons2.root"})};
auto muons     = {Create({"myMuons", "muons1.root"}),         Create({"myMuons", "muons2.root"})};

auto electronChain = CreateChain(electrons);
auto muonChain     = CreateChain(muons);

auto processor = CreateJoin({electronChain, muonChain}, {"run", "event"});
```

Join-first approach




```
auto emPair1 = {Create({"myElectrons", "electrons1.root"}), Create({"myMuons", "muons1.root"})};
auto emPair2 = {Create({"myElectrons", "electrons2.root"}), Create({"myMuons", "muons2.root"})};

auto electronMuonJoin1 = CreateJoin(emPair1, {"run", "event"});
auto electronMuonJoin2 = CreateJoin(emPair2, {"run", "event"});

auto processor = CreateChain({electronMuonJoin1, electronMuonJoin2});
```

Foreseen integration with RDataFrame

- The **RNTupleProcessor** will become the under-the-hood processing engine in **RDataFrame**;
  - ➔ Completely transparent to users!
- Opportunity to further evolve **RDataSetSpec**.
  - ➔ This will become the interface for “building” the **RNTupleProcessor**.

```
{
  "samples": [
    {
      "identifier": "electrons",
      "name": "myElectrons",
      "files": ["electrons1.root",
                "electrons2.root"],
      "joinWith": {
        "sample": "muons",
        "joinOn": ["run", "event"],
        "eventAlignment": "file"
      },
    },
    {
      "identifier": "muons",
      "name": "myMuons",
      "files": ["muons1.root",
                "muons2.root"]
    }
  ]
}
```

spec.json

Current status and outlook

- `RNTupleChainProcessor` and `RNTupleJoinProcessor` are currently available in ROOT master;
- Work towards enabling *composing* **RNTupleProcessor**s is well underway;
- Integration with **RDataFrame** is planned for 2025;
- Performance profiling and optimization for *unaligned joins* is planned for 2025;
- Support for the **RNTupleProcessor** in *distributed settings* (through **RDataFrame**) is foreseen.