Data set chains and joins with the RNTupleProcessor

Florine Willemijn de Geus, CERN EP-SFT & University of Twente (NL)
for the RNTuple development team

RNTuple Workshop 2024, CERN
December 1, 2024

Introduction

N.B. this presentation is adapted from my CHEP 2024 talk.

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

1. *Vertically* through the `TChain` interface;
2. *Horizontally* through the `TTree::AddFriend` interface, possibly using a `TTreeIndex` for unaligned entries.

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.

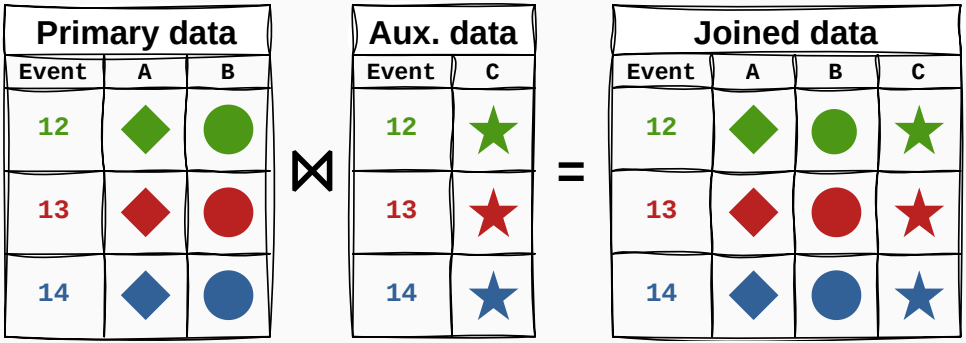
Use cases for data set joins

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

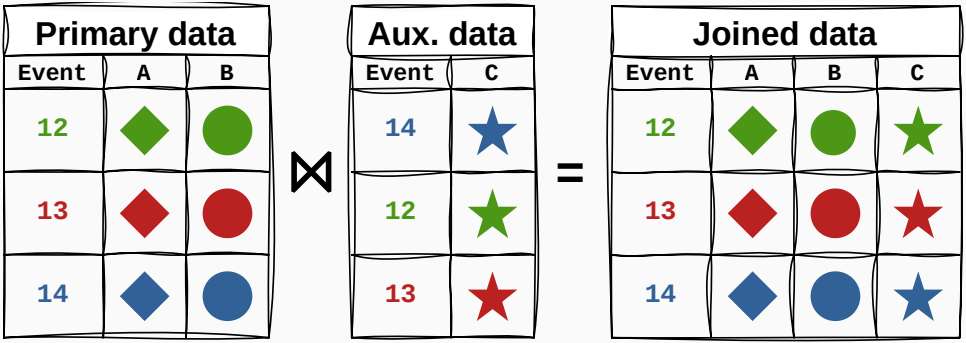
This would currently require copying the relevant fields from the central NanoAOD/PHYS(LITE)/… and these additional data into a custom **RNTuple**.

➔ (unnecessary) data duplication!

Data set joins: the ideal case

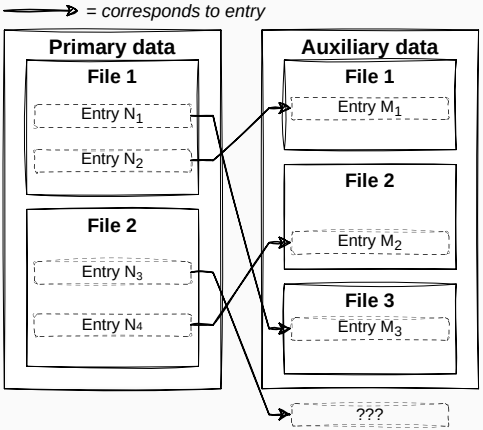


Data set joins: a realistic scenario



The caveats of unaligned data set joins

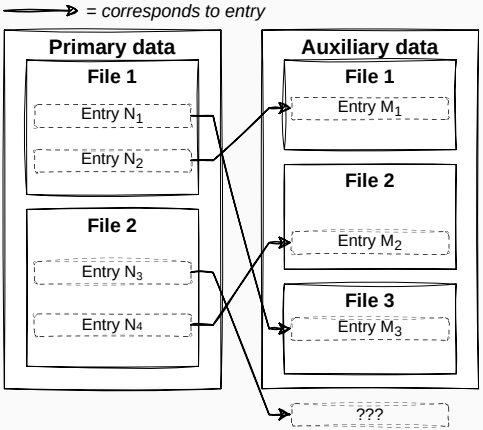
- Which events belong together?
  - Both false positives and negatives are unacceptable!
- What if the right-hand side event data is missing?
- What if my events are scattered across multiple files?
- What if want to distribute my analysis?



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?

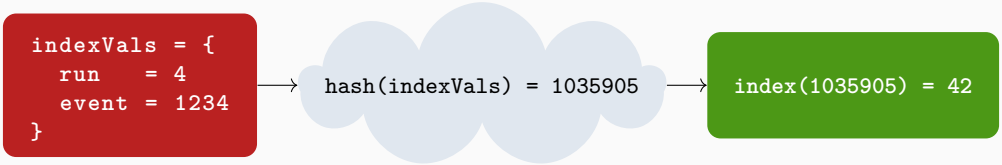
+ How to express all of this nicely?



Handling unaligned joins

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

- 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*



Our approach in **RNTuple**: current status

New data iteration model: `RNTupleProcessor`.

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

Our approach in **RNTuple**: current status

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

Our approach in **RNTuple**: current status

New data iteration model: `RNTupleProcessor`.

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

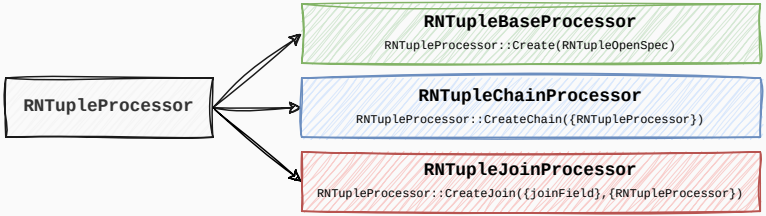
```
std::vector<RNTupleSourceSpec> ntuples{
    {"myElectrons", "electrons.root"}, {"myMuons", "muons.root"}};
auto processor = RNTupleProcessor::CreateJoin(ntuples, {"run", "event"});

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

Our approach in **RNTuple**: next steps

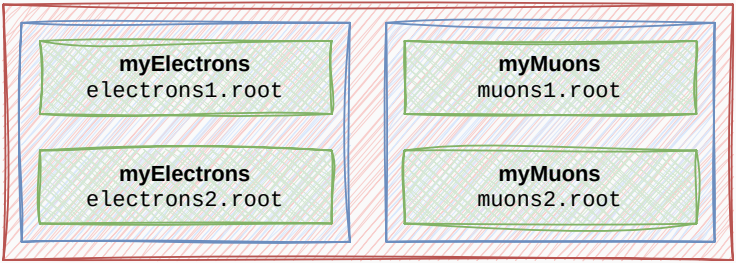
Make data sets fully composable:



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

This could potentially help speed up processing.

Chain-first approach

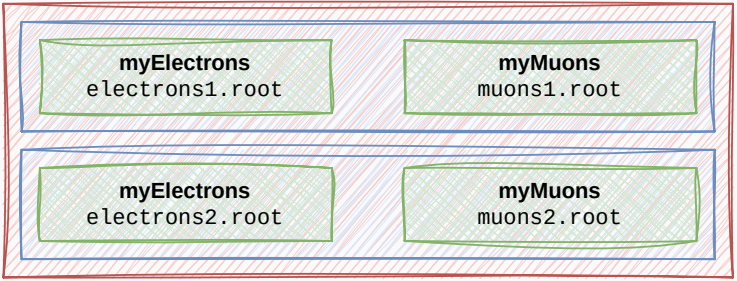



```
auto electrons = {Create({"myElectrons", "electrons1.root"}), Create({"myElectrons", "electrons2.root"})};
auto muons     = {Create({"myMuons", "muons1.root"}),         Create({"myMuons", "muons2.root"})};

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});
```

Performance considerations

Joining datasets will not come for free (especially when chains are involved).

→ Biggest bottleneck: building and **probing** the join index.

The cost of joining depends on:

- Number of events;
- Contents of the index values;
- “Scatteredness” of events.

Performance considerations

Joining datasets will not come for free (especially when chains are involved).

→ Biggest bottleneck: building and **probing** the join index.

Foreseen optimizations from our side:

- Tailor the join index to enable efficient multithreading;
- Ensure good distribution of hashed index values;
- Use on-disk data statistics to prevent unnecessary lookups.

N.B. The focus so far has been on the interface design – once this has been consolidated, performance will be addressed.

Performance considerations

Joining datasets will not come for free (especially when chains are involved).

→ Biggest bottleneck: building and **probing** the join index.

Help (where possible) from the domain experts:

- Guarantees when events will be aligned;
- Guarantees when events will be ordered;
- Hints which files belong together.

Foreseen integration with RDataFrame

```
{
  "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

```
df = ROOT.RDF.FromSpec("spec.json");
df_cuts = df.Filter("electrons.size >= 2 && muons.size >= 2")
            .Filter("goodPts(electrons.pt, muons.pt)")

df_mass_e = df_filtered.Define(
    "electron_mass",
    "InvariantMass(electrons.pt, electrons.eta, \
                   electrons.phi, electrons.mass)"
)
hist_mass_e = df_mass_e.Histo1D("electron_mass")


df_mass_e = df_filtered.Define(
    "muon_mass",
    "InvariantMass(muons.pt, muons.eta, \
                   muons.phi, muons.mass)"
)
hist_mass_m = df_mass_m.Histo1D("muon_mass")
```

analysis.py

Discussion starters

- Does this approach address all (or at least most) of the (currently known) use cases?
- Is this something that will mainly be used for analysis, or could it also have a place in core software frameworks?
- Any particular requirements for sparse or chunked reading?
- Would it make sense to split the composition interface from the processor interface?
  - ➔ i.e., a seperate `RNTuple[Base|Chain|Join]Composer` and `RNTupleProcessor`.
- Can (and if so how) can we check that ntuples semantically belong together?
- At some point we will have to address the persistification of the `RNTupleIndex`. This is still one of the biggest unknowns – any input/requirements/wishes are welcome!