On-the-fly data set joins and concatenations with ROOT’s RNTuple

Florine Willemijn de Geus[1,2] Vincenzo Eduardo Padulano[1]
Jakob Blomer[1] Philippe Canal[3] Ana-Lucia Varbanescu[2]

[1]CERN EP-SFT, Geneva (CH) [2]University of Twente, Enschede (NL) [3]FermiLab, Chicago (US)

CHEP 2024, Kraków, Poland
October 23, 2024
florine.de.geus@cern.ch

UNIVERSITY OF TWENTE.

Introduction

RNTuple is ROOT's next-generation columnar I/O subsystem, aiming at:

1. Higher storage space efficiency and lower CPU usage
2. Robust and modern interfaces
3. Efficient use of modern hardware and object stores

Adoption in experiment framework is well underway ▸ LHCb ▸ ATLAS ▸ CMS

Testing, evaluation and optimization is ongoing in anticipation of the first production release ▸ Plenary ▸ Direct I/O

At the same time, we can start looking at **RNTuple's role in complex data analysis use cases**

A typical HEP processing pipeline

$\mathcal{O}(MB)$ → $\mathcal{O}(kB)$

Event size

Detector data

MC data

Reconstruction

AOD

*Analysis Object Data

Further processing, skimming, ...

AOD'

Event selection, statistics, ...

Custom NTuple

Final plot

A typical HEP processing pipeline

$\mathcal{O}(MB)$ $\mathcal{O}(kB)$

Event size

Detector data

MC data

Reconstruction

AOD

*Analysis Object Data

Further processing, skimming, ...

AOD’

Event selection, statistics, ...

Custom NTuple

Final plot

Compact event data formats

CMS: NANOAOD [1]

- 1-2 kB per event
- Covering 50-70% of physics analyses
- Available for Run 2 data and beyond

ATLAS: PHYSLITE [2]

- 10-12 kB per event
- Covering up to 80% of analyses
- Prototype available for Run 3 data, projected to become default for Run 4

Both formats are pre-calibrated and can be used directly for analysis...

The need for data set joins

…But sometimes we need (or want) more data:

The need for data set joins

...But sometimes we need (or want) more data:

1. Analysis may require objects not present in the compact data format

The need for data set joins

…But sometimes we need (or want) more data:

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

Data set joins: the ideal case

Primary data

Event	A	B
12	◆	●
13	◆	●
14	◆	●

⋈

Aux. data

Event	C
12	★
13	★
14	★

=

Joined data

Event	A	B	C
12	◆	●	★
13	◆	●	★
14	◆	●	★

Data set joins: a realistic scenario

Primary data

Event	A	B
12	◆	●
13	◆	●
14	◆	●

⋈

Aux. data

Event	C
14	★
12	★
13	★

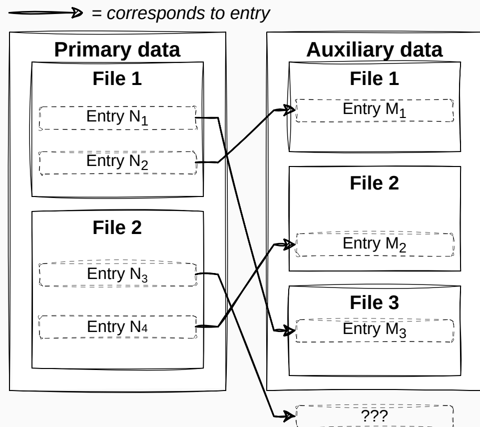
=

Joined data

Event	A	B	C
12	◆	●	★
13	◆	●	★
14	◆	●	★

The caveats of unaligned data set joins

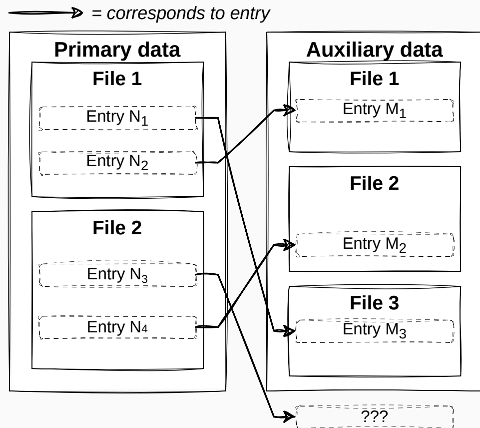
- Which events belong together?
  - Both false positives and negatives are unacceptable!
- What if the right-hand side event data is missing?
- What if my events are scattered across multiple files?
- What if want to distribute my analysis?



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?

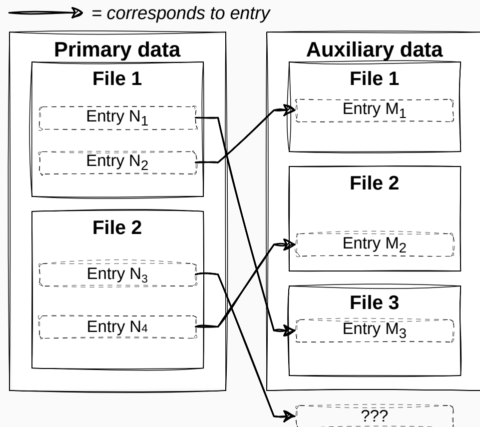
+ How to express all of this nicely?



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?



RNTuple's predecessor, TTree, addresses this through **friends**, combined with **chains**

Our end goal

```json
{
  "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

```python
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

Our approach in RNTuple

New data iteration model: **`RNTupleProcessor`**

Responsible for handling vertical concatenations and joins, in a unified way

Our approach in RNTuple

```
"files": ["electrons1.root",
          "electrons2.root"],
```

spec.json

New data iteration model: **`RNTupleProcessor`**

Responsible for handling vertical concatenations 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;
}
```

→ Available in ROOT master

Our approach in RNTuple

```
"joinWith": {
  "sample": "muons",
  "joinOn": ["run", "event"],
  "eventAlignment": "file"
},
```

spec.json

New data iteration model: `RNTupleProcessor`

Responsible for handling vertical concatenations and joins, in a unified way

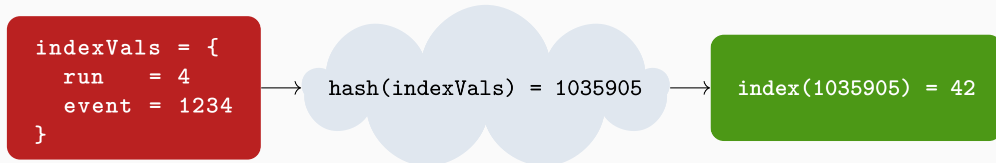
```
std::vector<RNTupleSourceSpec> ntuples{
    {"myElectrons", "electrons.root"}, {"myMuons", "muons.root"}};
auto processor = RNTupleProcessor::CreateJoin({"run", "event"}, ntuples);

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;
}
```

→ Under review (PR #16708)

The internals of 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*



Putting it together

```
std::vector<RNTupleSourceSpec> electronNTuples{
    {"myElectrons", "electrons1.root"}, {"myElectrons", "electrons2.root"}};
std::vector<RNTupleSourceSpec> muonNTuples{
    {"myMuons", "muons1.root"}, {"myMuons", "muons2.root"}};

auto electronProcessor = RNTupleProcessor::CreateChain(electronNTuples);
auto muonProcessor = RNTupleProcessor::CreateChain(muonNTuples);

auto processor = RNTupleProcessor::CreateJoin({"run", "event"},
    std::move(electronProcessor), std::move(muonProcessor));

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;
}
```

The cost of joins

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

The cost of joins

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

The cost of joins

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 (you!):

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

Summary and outlook

The **RNTupleProcessor** will enable non-trivial data access patterns, no data set duplication or external tools required

Major efforts are made towards minimizing the additional overhead of such operations

Seamless integration with (distributed) RDataFrame through a well-defined specification schema will be a key feature, expected in 2025

Have use cases or ideas not covered here? Reach out!

Find me during the coffee breaks, or drop me an email: `florine.de.geus@cern.ch`

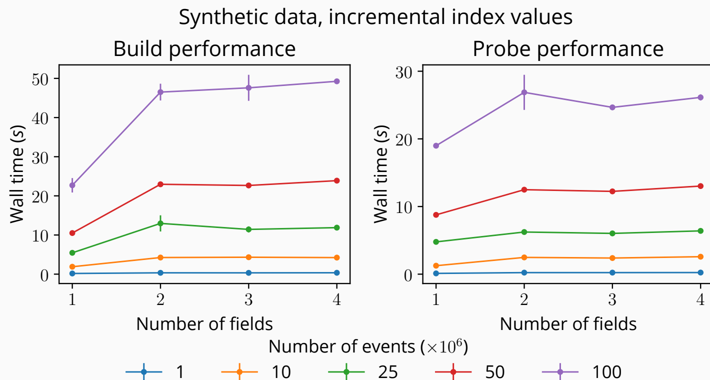
References

[1] Karl Ehatäht. "NANOAOD: A New Compact Event Data Format in CMS". In: *EPJ Web Conf.* 245 (2020), p. 06002. ISSN: 2100-014X. DOI: 10.1051/epjconf/202024506002.

[2] Jana Schaarschmidt et al. "PHYSLITE - A New Reduced Common Data Format for ATLAS". In: *EPJ Web of Conf.* 295 (2024), p. 06017. ISSN: 2100-014X. DOI: 10.1051/epjconf/202429506017.

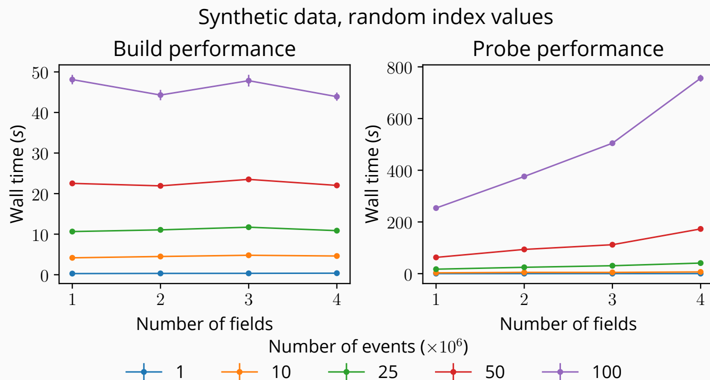
Backup

RNTupleIndex: first performance indication



These results are **highly** preliminary!

RNTupleIndex: first performance indication



These results are **highly** preliminary!