ROOT for future experiments

RDataFrame, RNTuple & more

Jakob Blomer (CERN) for the ROOT team

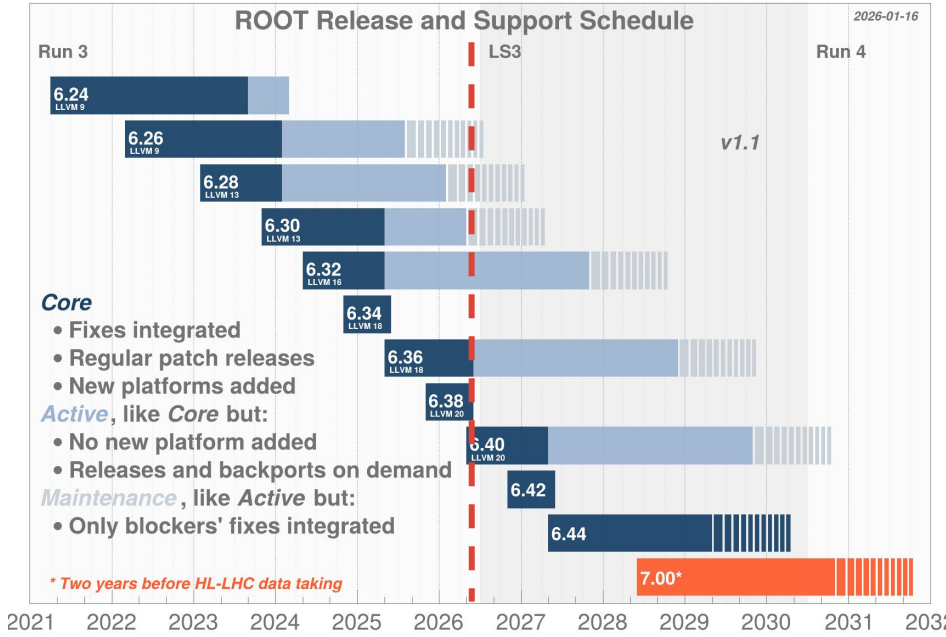
ePIC Software & Computing Weekly Meeting
10-06-2026

ROOT Roadmap

Primary Goals:

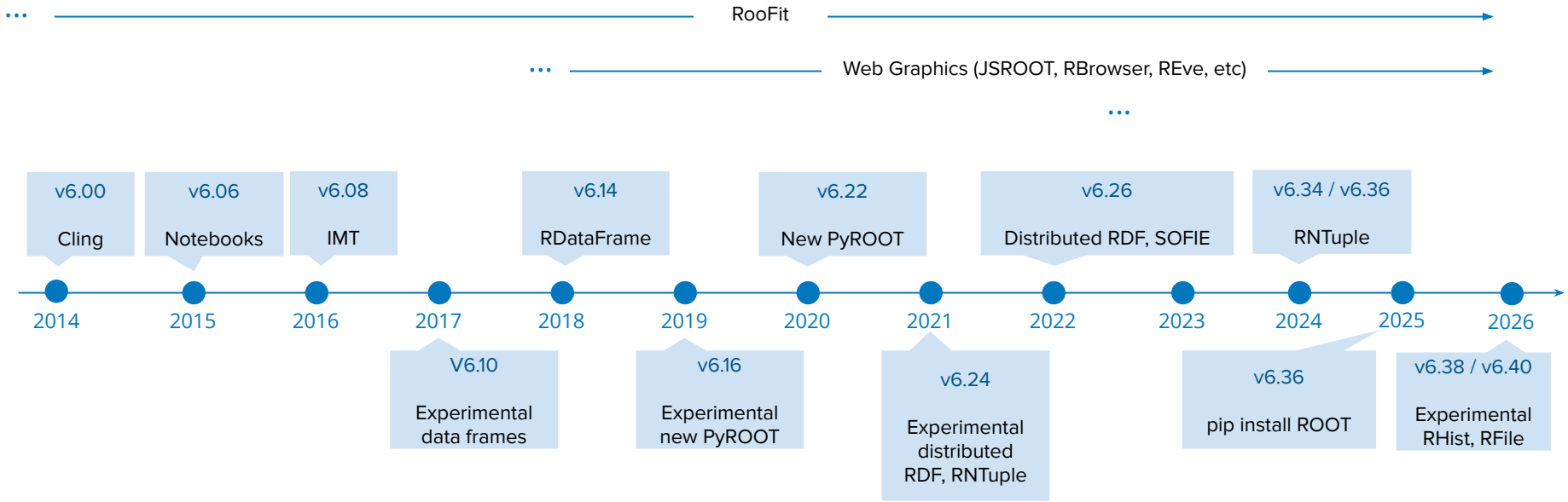
- **Focal point** for analysis tools & foundational data processing libraries
  - Integrated set of libraries and classes
- **Hub for R&D** activities
- Stability and maintenance **aligned with experiments life times**
- **Prioritise the experience from Python**
- **Simplify** packaging, installation, distribution
- **Tailor-made for HE(N)P**

Online on ROOT's website: https://root.cern/install/all_releases/



*Modelled based on a **suggestion by ePIC**!*

ROOT R&D, Selected Milestones

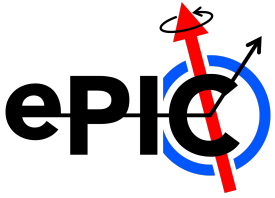


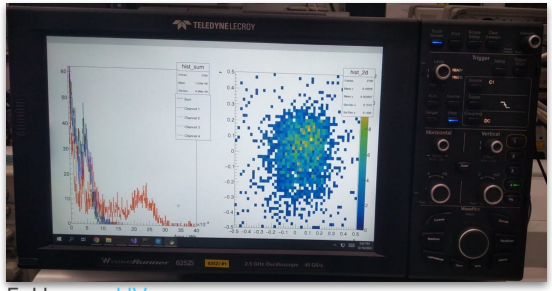
Track record of bringing multi-year R&D projects in production

ROOT 7: 2028 (2 years before Run 4)

- A piecewise renovation: expose new components, functionality and behaviours in ROOT6.
  - Some changes will be backward incompatible and
    will be accompanied by documentation and migration patterns
  - A small jump compared to ROOT5 → ROOT6, if new components are adopted early!
- Discussions will take place at an increasing frequency, starting this year, e.g. at the Parallelism, Performance and Programming model meetings: external input is welcome (and precious for ROOT)

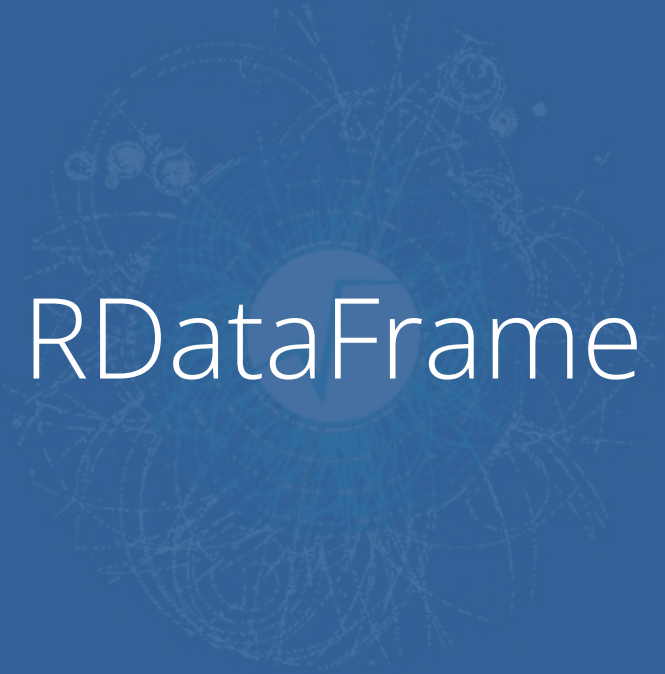
ROOT: More than the LHC Community!





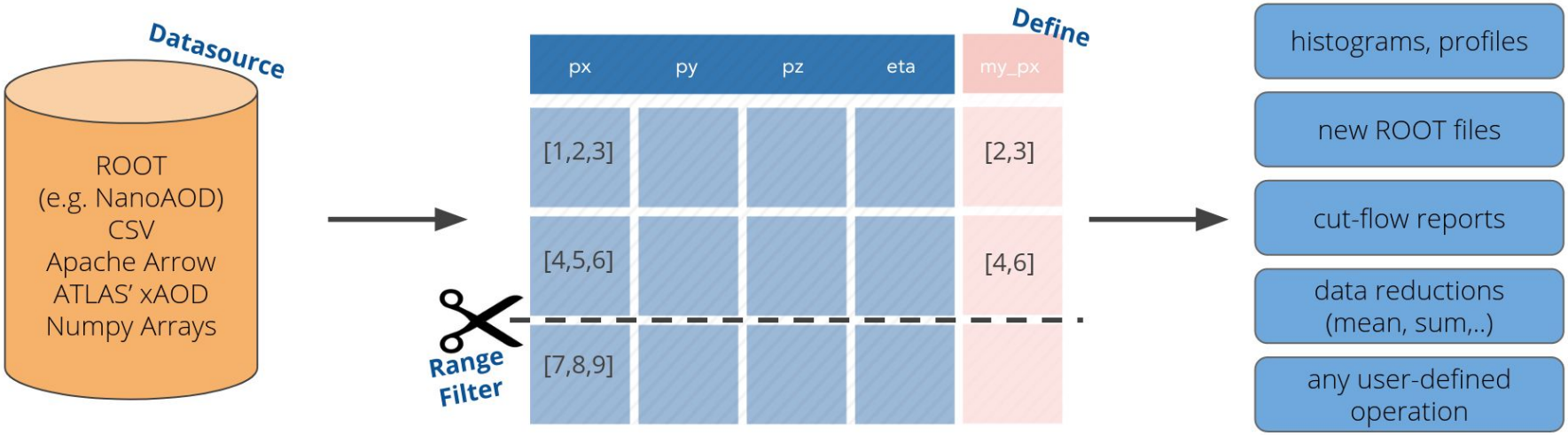
F. Hueso, UV

And much more…



RDataFrame

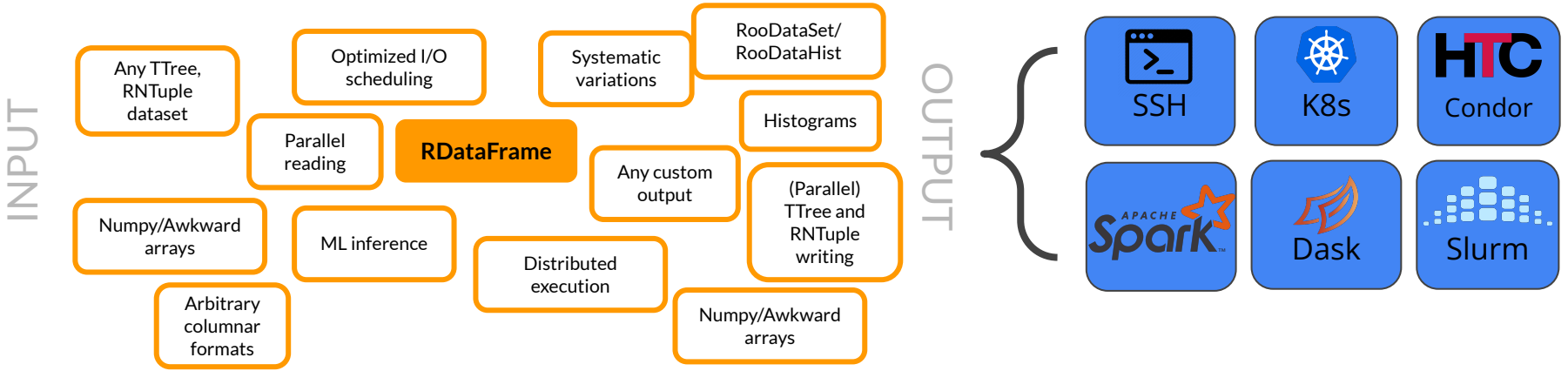
RDataFrame in a Nutshell



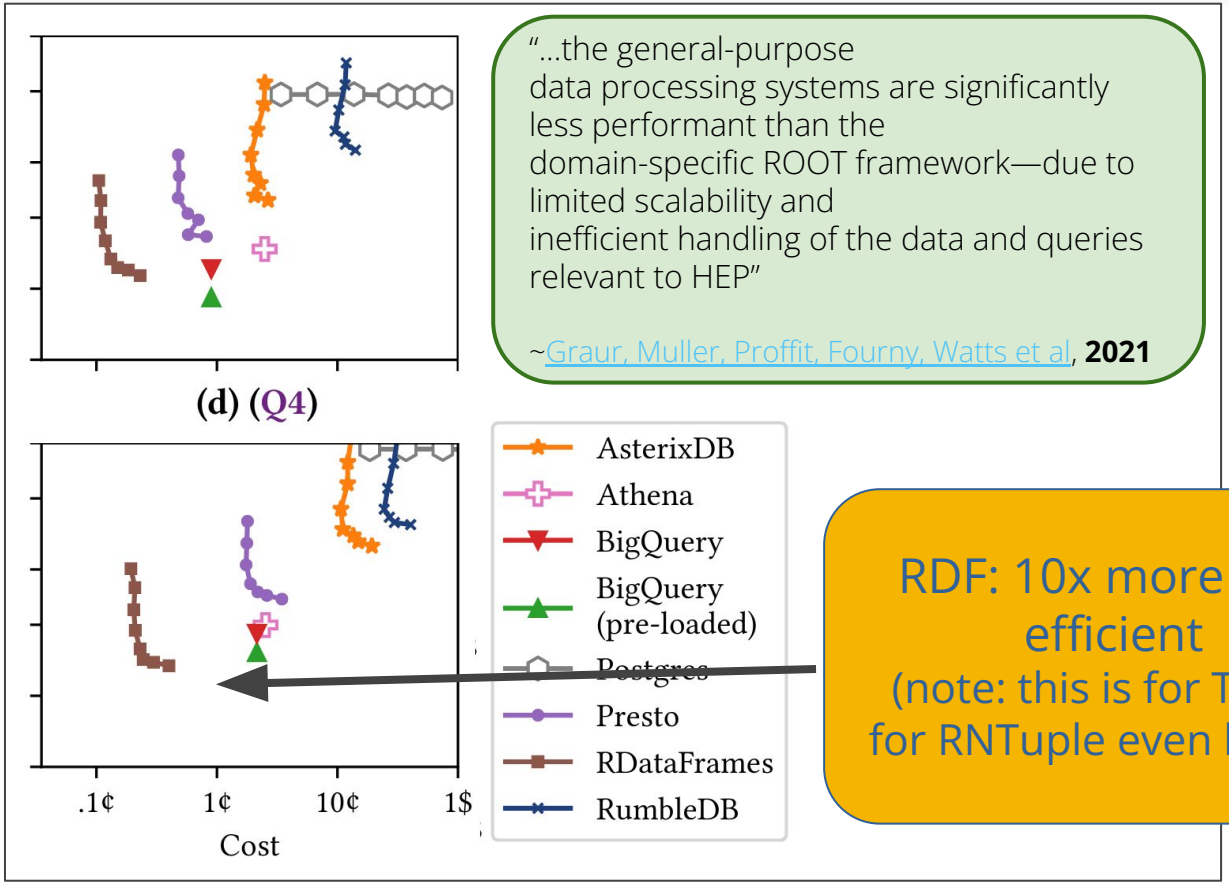

```
df = ROOT.RDataFrame(TTreeName/RNTupleName, fileNames)
histo = df.Define(...).Filter(...).Vary(...).Histo1D(...)
histo.Snapshot(...)
```

Entry Point to Modern ROOT

- ROOT's high-level interface for analysis
- Supports native parallel execution
- Multi-threaded and distributed



Tailor-made for HEP data processing



"...the general-purpose data processing systems are significantly less performant than the domain-specific ROOT framework—due to limited scalability and inefficient handling of the data and queries relevant to HEP"

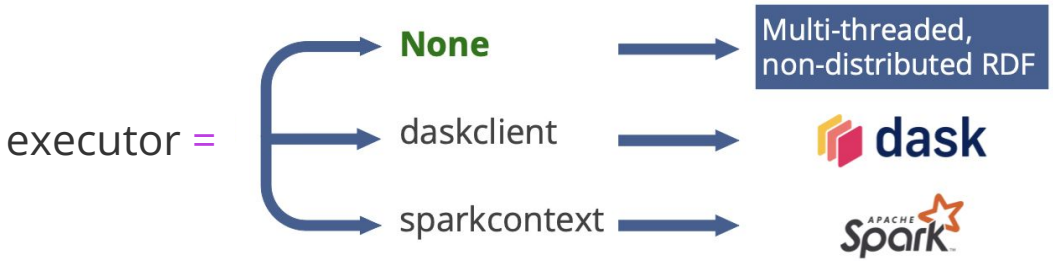
~Graur, Muller, Proffit, Fourny, Watts et al, **2021**

RDF: 10x more cost efficient
(note: this is for TTree, for RNTuple even better)

Distributed RDataFrame

- Distributed RDataFrame is in production
  - constructing distributed RDF became as simple as:

```
df = ROOT.RDataFrame(treeName/RNTupleName, fileNames, executor)
```

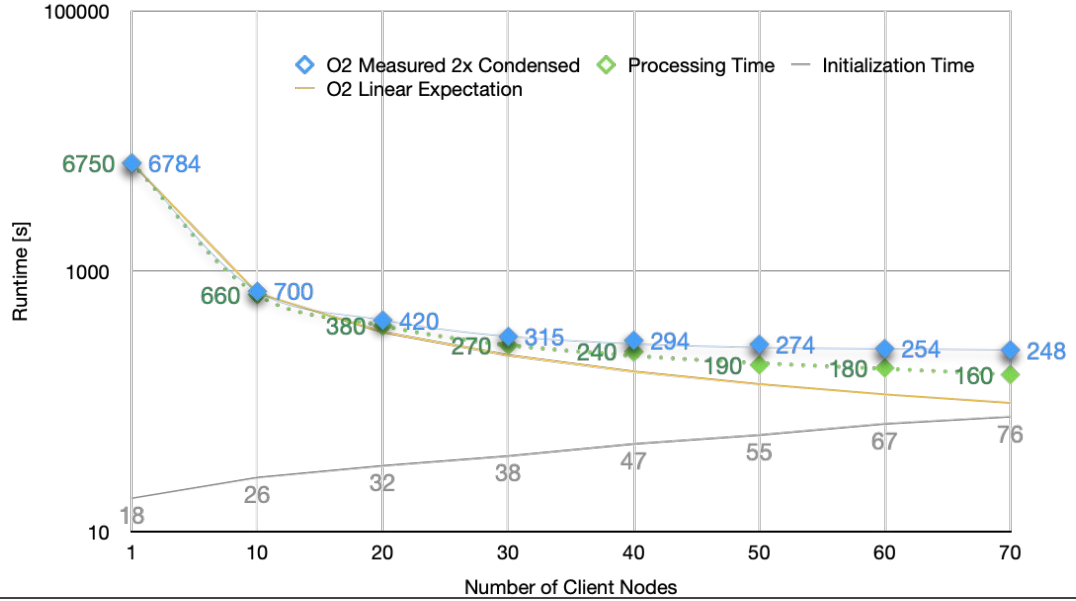


- Functionality-wise, distributed and local RDFs are (almost) equivalent

In production, at scale

RNTuple CHEP'24 Plenary
October 2024

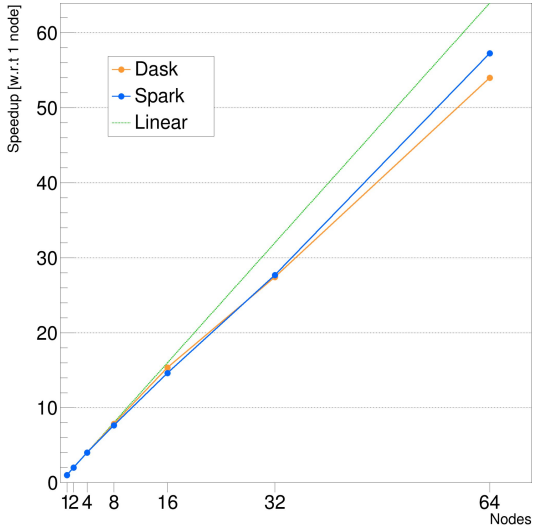
Single Analysis
reaches avg. INGRES
222 GBit/s
during processing
345 GBit/s

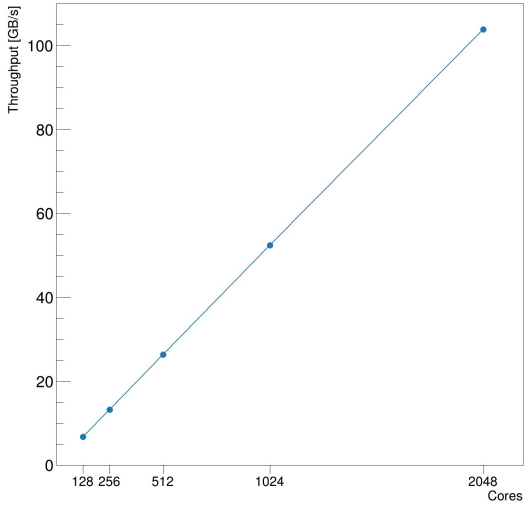


In production, at scale

CERN HPC
Slurm jobs (Spark/Dask)
~100 GB/s on 2048 cores
JGC publication, February 2023

Jülich HPC
Collaboration with OpenLab
Slurm jobs (via Dask)
Presentation, August 2023

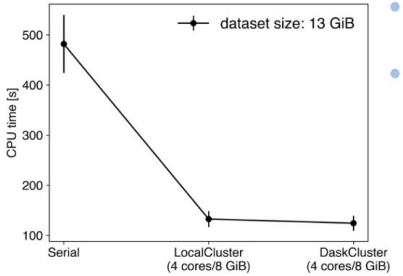




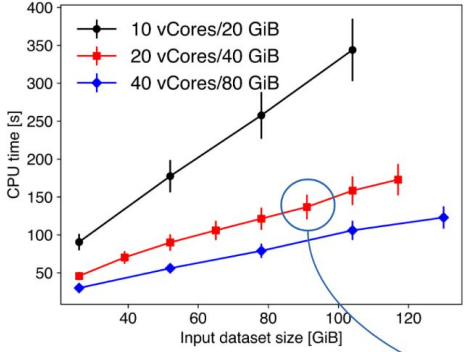
...with great performance

Finanziato dall'Unione europea NextGenerationEU | Ministero dell'Università e della Ricerca | Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA | ICSC Centro Nazionale di Ricerca in HPC, Big Data and Quantum Computing

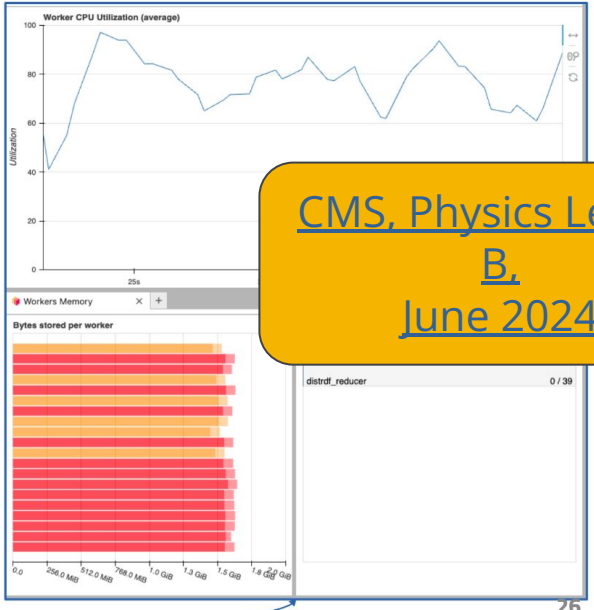
Preliminary results



- Significant improvement in execution time *wrt* the standard/serial approach
- The facility allows for dynamically scaling the resources, here testing the performance at fixed #cores and memory, varying the dataset size



- Stress test at high CPU and memory occupancy
- Stable performance, linearly scaling with the input dataset size
- Dataset size ~ 100 GiB is representative of ~15 /fb of Run3 data for this specific analysis



[CMS, Physics Letters B, June 2024]

26

In production, at scale

Giovanni Petrucciani, PPP174
December 2024

The analysis used for these tests

- A complete CMS analysis on run 2 + 2022 data, at full complexity
  - correctionlib + CMSJMECalculator for applying scale factors, calibrations, uncertainties, …
  - Onnx C++ runtime to evaluate DNNs from RDataFrame
- Measurements in a rare final state (top + Z processes), so only a modest total number of events after preselection, but from many different datasets
  - Often needing different computation graphs depending on data taking era, MC sample, …

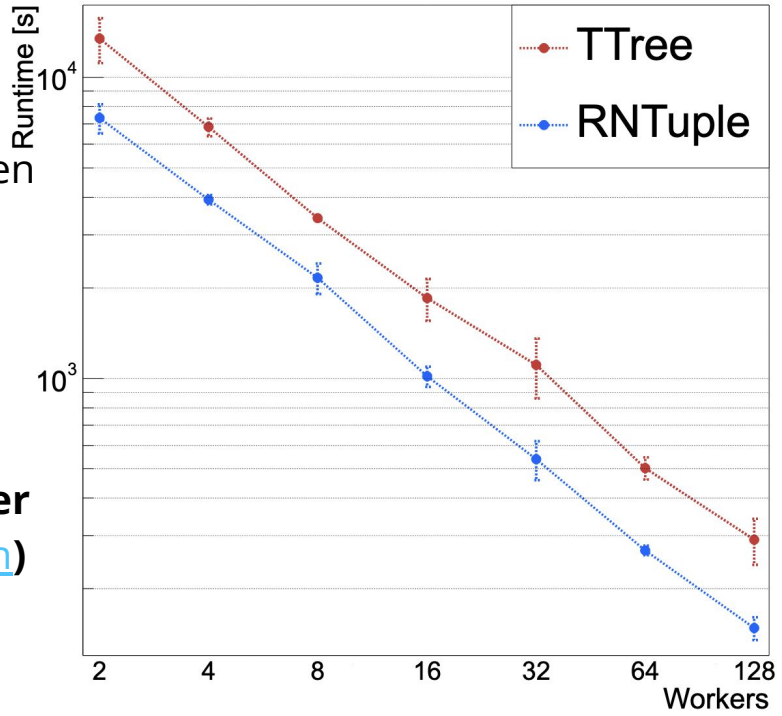
Dataset	Graphs (unique)	JITed functions	Files	Evts	Size
Just 2018	70 (43)	138	63	66 M	214 GB
Full dataset (2016-2022)	412 (248) (x 5.8)	260	362 (x 5.7)	273 M (x 4.1)	977 GB (x 4.6)

In production, at scale

[The Analysis Grand Challenge]

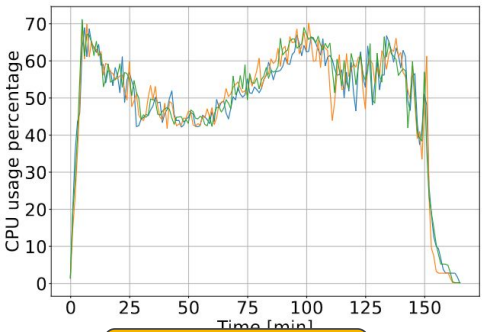
[RDF implementation]: **collaboration** between **ROOT** and **IRIS-HEP** [1],[2]

Linear scaling on SWAN
RNTuple always **faster** (CHEP'24 presentation)

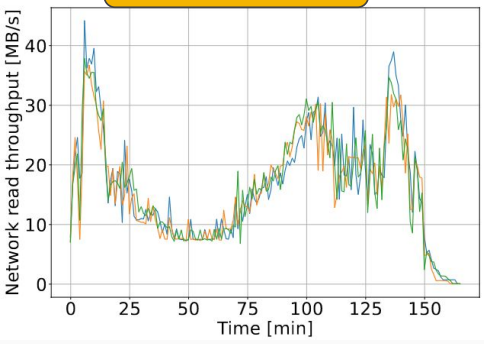


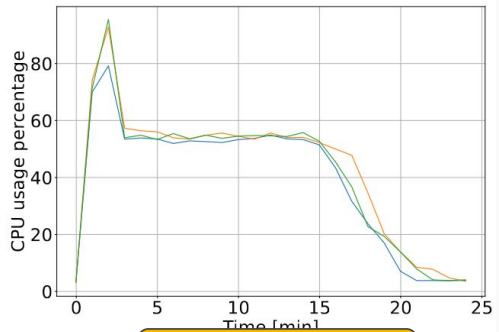
In production, at scale

- **CMS** analysis used on **Analysis Facilities**
- Before: Python **for-loop** with NanoAODtools, **manual** job submission
- After: **Interactive** distributed RDataFrame
- **O(10) speedup**
- Publication – February 2024

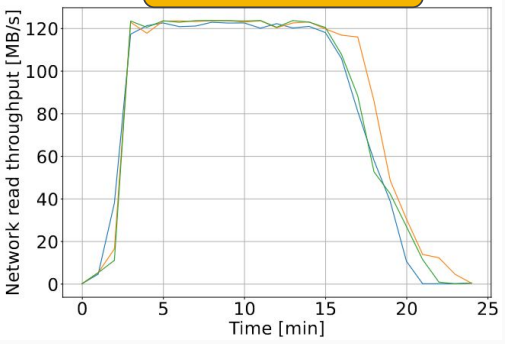


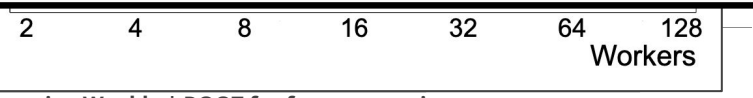
Legacy





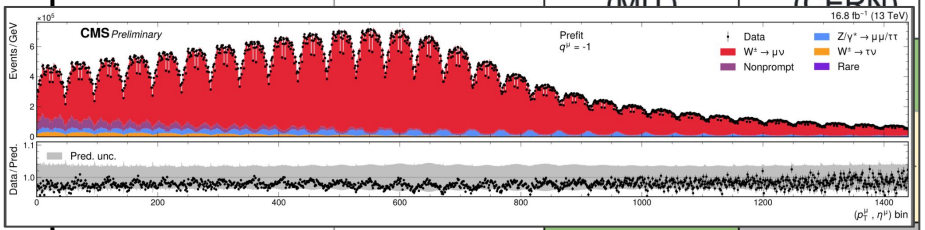
Distributed RDF





In production, at scale

CMS W Mass analysis with RDataFrame



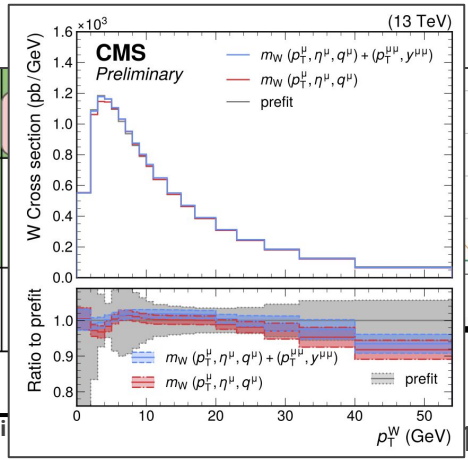
Summary

Increasing amount of data opens new opportunities

- Software developments must be ahead to fully exploit potential

Fast analysis turnaround was essential for this complex measurement

- RDF provides a convenient and efficient library
  - Initially showstoppers observed in scaling
  - Extensive work on critical parts to improve RDF and histogram implementation
- Full analysis runs in ~hours



ROOT RDataFrame

RDataFrame → ~1-2 h → Boost Histograms

Critical parts in c++

- Functions: e.g. check if reco muon has a match to any gen muon

```
bool hasMatchDR2(const float& eta, const float& phi, const Vec_f& vec_eta, const Vec_f& vec_phi, const float dr2 = 0.09) {

  for (unsigned int jvec = 0; jvec < vec_eta.size(); ++jvec) {
    if (deltaR2(eta, phi, vec_eta[jvec], vec_phi[jvec]) < dr2) return true;
  }
  return false;

}
```

xcache	not planned
SKGateway + Condor	DASKGateway + K8s

In production, at scale

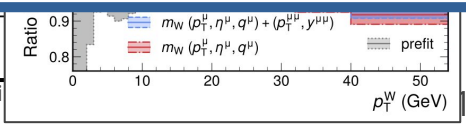
CMS W Mass analysis with RDataFrame

Conclusions

J. Gooding - RDataFrame in LHCb Offline Selection, 26.10.23

- RDataFrame provides **notable speedup** of offline selection, *if implemented optimally:*
  - *Potential speedup of* $\mathcal{O}(10)$ in some cases.
- uproot+numba **matches RDataFrame performance** in many scenarios:
  - I/O bottleneck identified → performance improvement possible.
- Both tools suitable for **fast parallelised selection**, necessary for upcoming Run 3 analyses.


```
    return false;
}
```



In production, at scale

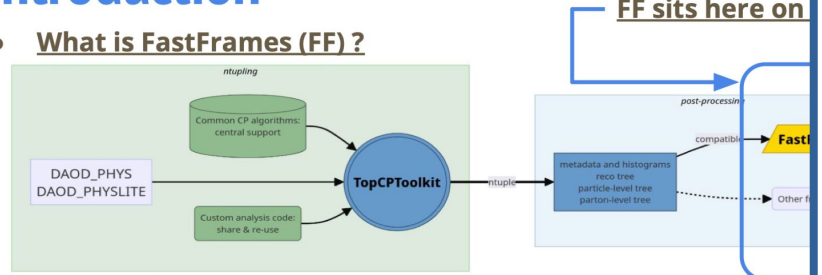
T. Dado, ATLAS FastFrames

FastFrames performance

- **Tested with a realistic example**
  - TTbar single lepton sample with ~ **20M events** (200M events in the DAOD)
    - ~ 400 GBs
  - Producing many control histograms
  - Producing analysis specific histograms
  - Running on all (automatic) systematics: ~400 systematics
  - Total of >25k histograms (total in three regions)
- **Ran on Chicago analysis facility**
  - systematics run, **25k+ histograms**, 10 GB ram requested, **10 CPUs**, running time: **10h 12m**
  - systematics run, **25k+ histograms**, 20 GB ram requested, **50 CPUs**, running time: **2h 52m**
  - systematics run ~**20k histograms**, 20 GB ram requested, **50 CPUs**, running time: **2h 20m**
  - nominal only run, **60 histograms**, 20 GB ram requested, **50 CPUs**, running time: **< 2m**
- **HTCondor**
  - nominal only run, ~**4k histograms**, ~**250 regions**, **30 CPUs**, running time: **< 16 m**
    - ~ **66M events**

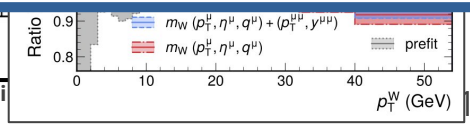
Introduction

- **What is FastFrames (FF) ?**



  - Is a framework to produce **histograms/ntuples**.
  - Compatible with the outputs produced by frameworks using **ATLAS format**
  - Build around **ROOT::RDataFrame** technology.
- **What can FF do ?**
  - Run in Linux and MacOS.
  - NTuples → **NTuples.**
  - NTuples → **Histograms.**
    - Supports 1D, 2D, 3D and TProfile
  - Process **reco and truth trees** – including event matching.
  - Prepare inputs for **unfolding** – which can be passed to TRExFitter directly.

ary for upcoming Run 3 analyses.



Systematic Variations

Declare column to vary and Vary expression

Variation labels

More observables depending on the varied values with automatic bookeeping

```
nominal_hx = (
    df.Vary(
        "pt",
        "{pt*0.9, pt*1.1}",
        ["down", "up"])
      .Filter("pt > k")
      .Define("x", someFunc, ["pt"])
      .Histo1D("x")
)
hx =
ROOT.RDF.Experimental.VariationsFor(nominal_hx)
hx["nominal"].Draw()
hx["pt:down"].Draw("SAME")
```

Dictionary of varied histograms

Systematic Variations

- Declare variation expression and labels
- **Automatically propagate** through the computation graph
- In **this** case produces **three universes**: `nominal`, `pt:up`, `pt:down`

```
nominal_hx = (
    df.Vary(
        "pt",
        "{pt*0.9, pt*1.1}",
        ["down", "up"])
      .Filter("pt > k")
      .Define("x", someFunc, ["pt"])
      .Histo1D("x")
)
hx =
ROOT.RDF.Experimental.VariationsFor(nominal_hx)
hx["nominal"].Draw()
hx["pt:down"].Draw("SAME")
```

Snapshot with systematic variations

- Let's take an RDF where column x gets varied
- y depends on x, so has to vary accordingly

Writing

```
auto h = ROOT::RDataFrame(N)
            .Define("x", [](ULong64_t e) -> float { return 10.f * e; }, {"rdfentry_"})
            .Vary(
               "x", [](float x) { return ROOT::RVecF{x - 0.5f, x + 0.5f}; }, {"x"}, 2, "xVar")
            .Define("y", [](float x) -> double { return -1. * x; }, {"x"})
            .Filter(cuts, {"x", "y"})
            .Snapshot("t", filename, {"x", "y"}, options);
```

```
ROOT::RDF::RSnapshotOptions options;
options.fOverwriteIfExists = true;
options.fIncludeVariations = true;
```

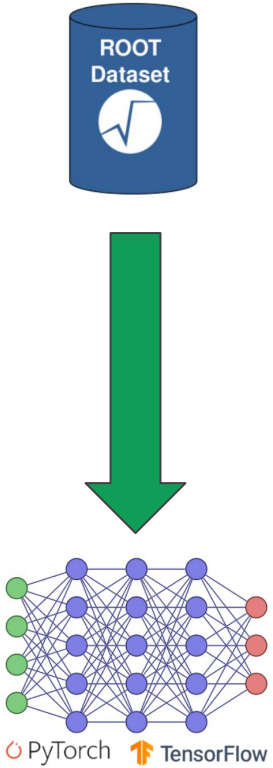
- When storing to disk, even though only x and y are requested, their variations are automatically included

Zero-overhead ML training with ROOT

- Go from ROOT to ML training without intermediate files
- Leverage all the power of RDataFrame and ROOT's I/O
- Prepare batches in a background thread, **optimally reading the ROOT file**
- Feed directly to the most popular ML frameworks

👉 Optimizations and New Strategies for Native ROOT Data Loading for ML

👉 Reverse path for inference: SOFIE



DataLoading tutorials

```python
import ROOT

rdataframe = ROOT.RDataFrame(....)
rdataframe = rdataframe.Filter(...)

train = ROOT.Experimental.ML.RDataLoader(
  rdataframe,
  batch_size=128,
  batches_in_memory=10,
  target="label",
)

for x, y in train.as_torch(device="cuda"):
  model.train(x, y)
```

RDataFrame Pythonic API

👉 Arbitrary Python Execution in C++

- Use Numba to JIT-compile Python functions directly into the RDataFrame event loop

```
def invariant_mass(px, py, pz, E):
    return np.sqrt(E**2 - (px**2 + py**2 + pz**2))

rdf = ROOT.RDataFrame("tree", "file.root").Define("mass", invariant_mass)
```

💡 Directly pass your Python function to `Define` and `Filter`

```
@ROOT.Numba.Declare(["ROOT::Math::PtEtaPhiMVector"], "float")
def get_mass(v):
    return v.M()

rdf = rdf.Define("m", "Numba::get_mass(v)")
```

💡 For ROOT types, use a decorator

💡 The decorator generates a wrapper you can use

RNTuple & I/O

Main I/O Components

`ROOT File`

Store and retrieve any object.
Organize objects in a directory hierarchy.
Local or remote access (XRootD, HTTP)

→ `TFile`

`TTree`

Special object in the ROOT file for efficient storage of event data

>2EB LHC Run 1–3 Data

`RNTuple`

Next generation columnar data storage

- *More compact*
- *Faster read and write*
- *Parallelization built-in*
- *Ready for object stores*
- *Robust API*

- Stable, self-contained on-disk format for decades
- Interface modernization underway: ➜RFile

- Focus of attention on the RNTuple development
- At the same time, significant efforts on the classic I/O
  - e.g. lift 1 GB limit
- TTree remains supported

RFile: like TFile, but easier

The basic operations are designed as a simplified version of TFile's:

	TFile	RFile
Opening	`auto file = unique_ptr<TFile>(TFile::Open("f.root"));`	`auto file = RFile::Open("f.root");`
Getting an object (owned)	`auto hist = unique_ptr<TH1D>(file->Get<TH1D>("hist"));` `hist->SetDirectory(nullptr);`	`auto hist = file->Get<TH1D>("hist");`

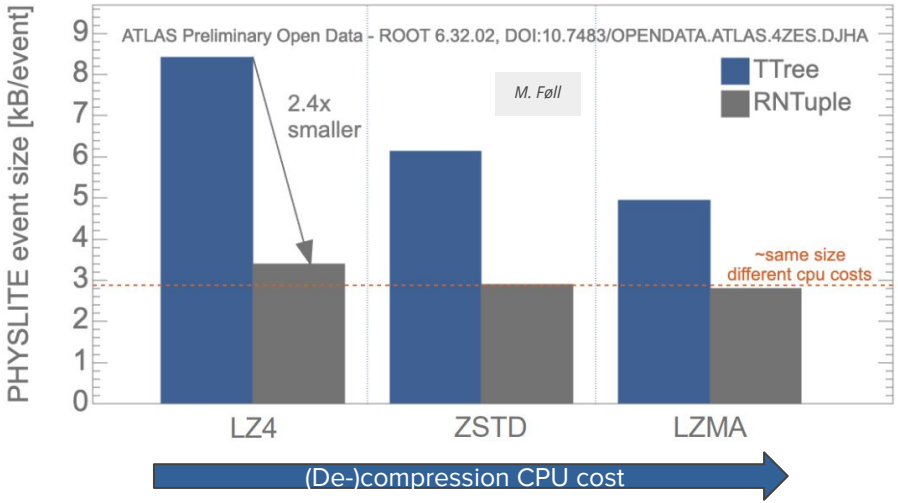
Putting an object	`file->WriteObject(hist, "hist");`
	`// or:`
	`file->WriteTObject(hist);`
	`// or:`
	`hist->Write();`
		`file->Put("hist", *hist);`

Handling errors when opening	`TFile *f = TFile::Open("f.root");`
	`if (!f || f->IsZombie()) {`
	`  // error`
	`}`
		`try {`
		`  auto f = RFile::Open("f.root");`
		`} catch (...) {`
		`  // error`
		`}`

RFile basic usage (Python)

```python
import ROOT

RFile = ROOT.Experimental.RFile

# Reading
with RFile.Open("file.root") as file:
  # From top-level dir:
  hist = file.Get("hist")
  # From sub dir:
  subhist = file.Get("my/path/hist")

# Writing
with RFile.Recreate("file.root") as file:
  # To top-level dir:
  file.Put("hist", hist)
  # To sub dir:
  file.Put("my/path/hist", hist)

# Querying
for key in file.ListKeys("my"):
  print(key.GetPath())
```

RNTuple Performance: Data Volume

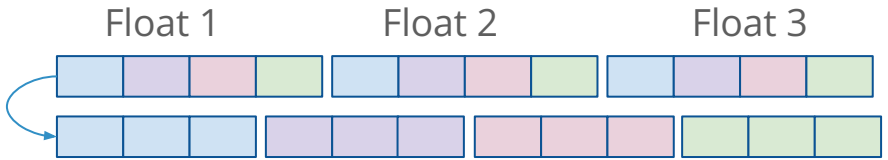


Example: ATLAS DAOD

Note that due to data preconditioning in RNTuple, the relative difference between compression algorithms fades.

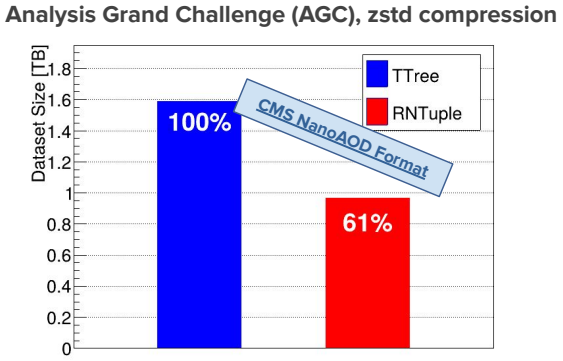
Contributors to space savings

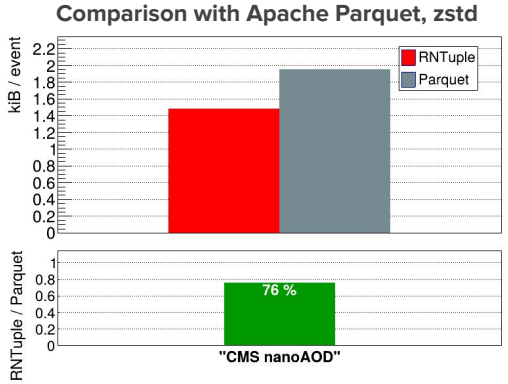
- More compact on-disk representation of collections and booleans (trigger bits)
- Same page merging
- Type-based data encoding optimized for better compression ratio
- Example: byte splitting

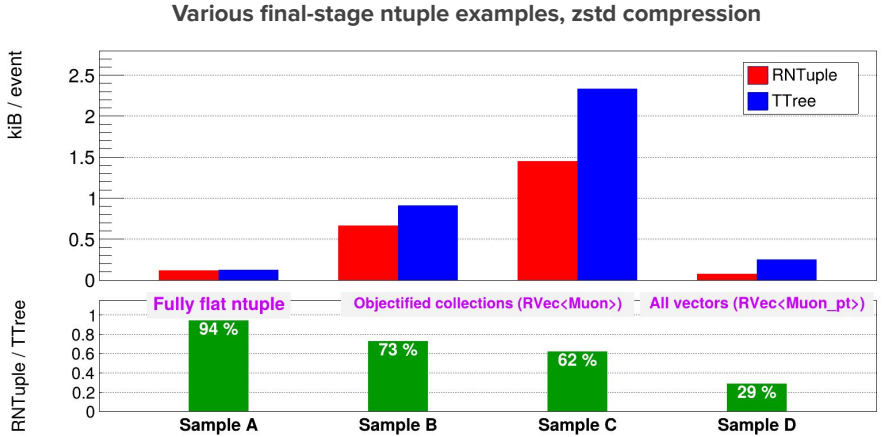


- Other encoding methods: delta encoding, zigzag encoding

RNTuple Performance: Data Volume



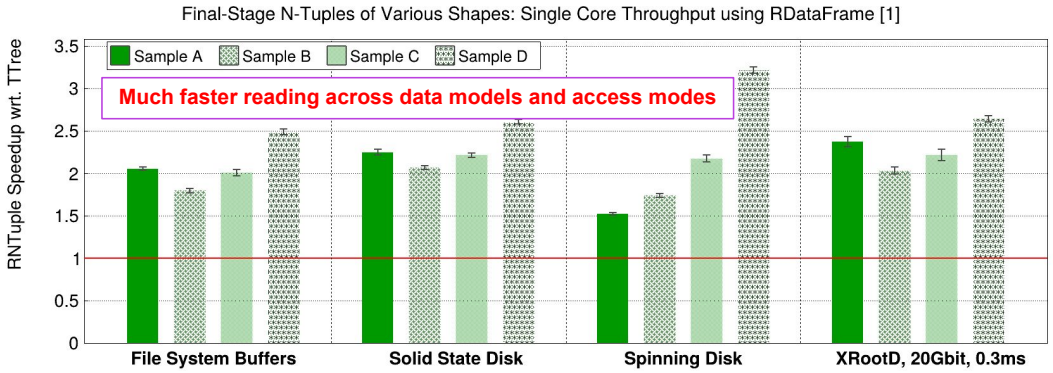




Ongoing investigation on compression ratio of some of the more upstream data products (Full AOD, RECO), which are currently in the ballpark of 10% – 15%

(we think that >=15% is still reachable even in these cases)

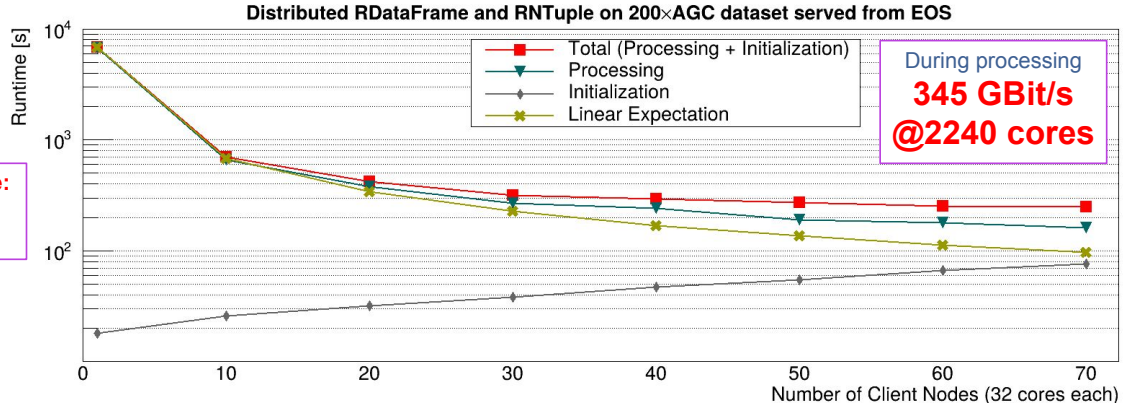
RNTuple Performance: Read Throughput



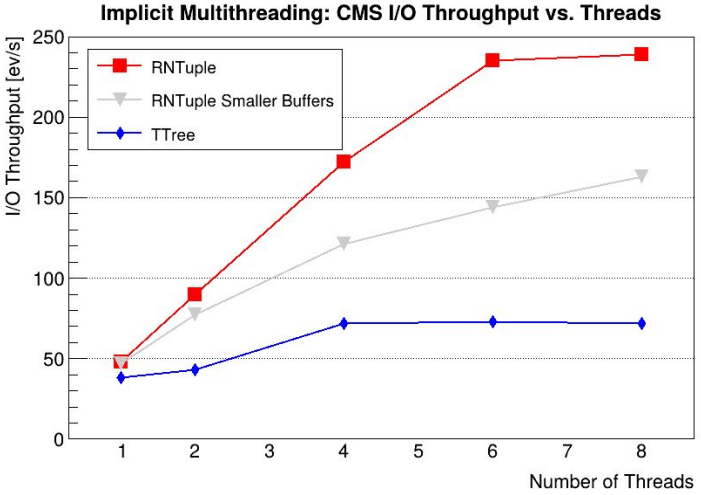
Contributors to higher throughput

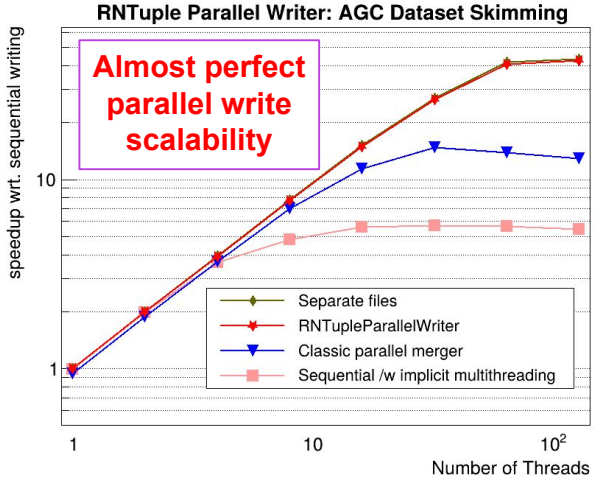
- Asynchronous prefetching
- Multi-stream disk access through `io_uring`
- Code optimization
- New on-disk layout allows for higher degree of explicit and implicit parallelization
- New RDataFrame I/O scheduler

Single-Node Performance: >3x faster than TTree (not plotted)



RNTuple Performance: Write Throughput

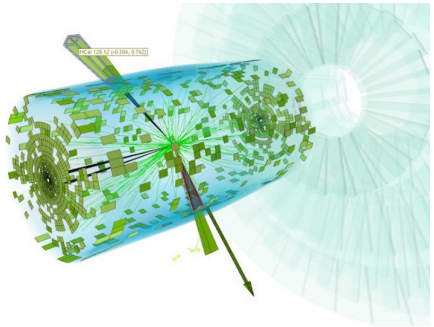




➜ https://cds.cern.ch/record/2919226/

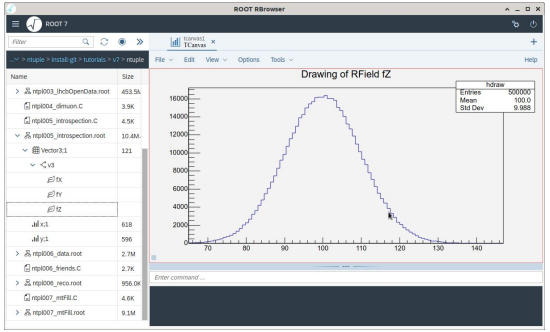
- Truly parallel writing; prototype support for multi-process and MPI support
- Capable of fully exploiting NVMe drives with ➜Direct I/O
- Reaching throughput values of several GB/s per node

Event Data in Practice



```
root [1] .ls
TFile**         /data/gg_data.root
 TFile*         /data/gg_data.root
  KEY: TTree    mini;55 mini [current cycle]
  KEY: TTree    mini;54 mini [backup cycle]
  KEY: ROOT::RNTuple    mini_imported;1
root [2]
```

A TTree and an RNTuple in the same ROOT file. In this example, the RNTuple data has been converted from the tree using the **`RNTupleImporter`**.



RBrowser showing RNTuple data

- As with TTree, RNTuple data
  - is stored in **ROOT files**
  - can be processed with **RDataFrame** without changing any code, including distributed RDF and snapshotting
  - can be explored in the browser: **RBrowser, TBrowser, *JSROOT*** *(currently only partial)*
  - can be merged using **TFileMerger** (including parallel merge mode) & **hadd**
  - is subject to → **I/O customization rules** and **automatic schema evolution**

Rich Event Data Models (EDMs)

Type Class	Types	EDM Coverage			RNTuple Status
PoD	`bool, char, std::byte, (u)int[8,16,32,64]_t, float, double`	Flat n-tuple	Reduced AOD	Full AOD / ESD / RECO	Available
Records	Manually built structs of PoDs				
(Nested) vectors	`std::vector, RVec, std::array,` C-style fixed-size arrays				Available
String	`std::string`				Available
User-defined classes	Non-cyclic classes with dictionaries				Available
User-defined enums	Scoped / unscoped enums with dictionaries				Available
User-defined collections	Non-associative collection proxy				Available
stdlib types	`std::pair, std::tuple, std::bitset, std::(unordered_)(multi)set, std::(unordered_)(multi)map`				Available
Alternating types	`std::variant, std::unique_ptr, std::optional`				Available
Streamer I/O*	All ROOT streamable objects (stored as byte array)				Available
Low-precision floating points	`Double32_t, f16`	*Optimization benefitting all EDMs*			Available
	Custom precision / range (bfloat16, TensorFloat-32, other AI formats)				Available

TTree and RNTuple - comparison

Feature	TTree	RNTuple
Schema evolution	Manual and automatic	Manual and automatic
Merging	Supported	Supported
Backfilling	Supported	Limited support (late model extension)
Multi-threading support	Limited support	Built in the design
Metadata	Limited support	Supported (to be stabilized in 6.42)
Dataset combinations	Supported (with caveats)	Supported (to be stabilized in 6.42)

TTree and RNTuple - EDM support

Type class	TTree	RNTuple
PODs and structs	Supported	Supported
Nested vectors/collections	Only 1 level of splitting	Supported, fully split
STL classes	Support for std::optional, std::variant planned	Well-defined list of supported classes
Raw pointers	Supported	No support
TString/TNamed/TClonesArray	Supported	No support
Dynamic polymorphism	Supported	No support
Intra-event references* (TRef)	Supported	No support

deprecated in modern EDMs

*****future extensions of RNTuple might add another form of intra-event references*

Recommendations for EDMs

- ✓ Nested collections are well supported (`std::vector<std::vector<T>>`)
- ✓ Prefer `std::vector` over sets/maps when possible
- ✓ `std::optional`, `std::variant` are available (may replace dynamic polymorphism or raw pointers in some cases)
- ✓ You can use `std::unique_ptr` for heap-allocated data
- ✓ Prefer `std::(u)int*_t` types to platform-dependant primitive/ROOT types (`int, long, Int_t, Long_t, …`)
- ✓ Use types matching the content, e.g. `std::byte` for binary data instead of `UInt_t`

- ✓ In general, keep it simple
  - Represent data how it makes logical sense
  - RNTuple will generally do a good job storing it in an optimized way
  - e.g. std::vector of `struct Muon { float px, py, pz; }` rather than separate vectors of floats `muon_px, muon_py, muon_pz`; let RNtuple split structs into columns for you.

Recommendations for EDMs

- ❌ Don't derive your classes from `TObject` or `TNamed`
- ❌ Don't use streamer fields (they're there as a transitional measure from TTree)
- ❌ Some typical "optimization patterns" that were valid for TTree became antipatterns/unsupported in RNTuple:
  - Leaf count arrays (use vectors of structs instead)
  - TClonesArray (use vectors instead)
  - Leaf lists (use structs instead)

```
class MyClass {
  int n;
  float *px; //![n]
  float *py; //![n]
  float *pz; //![n]
};
```

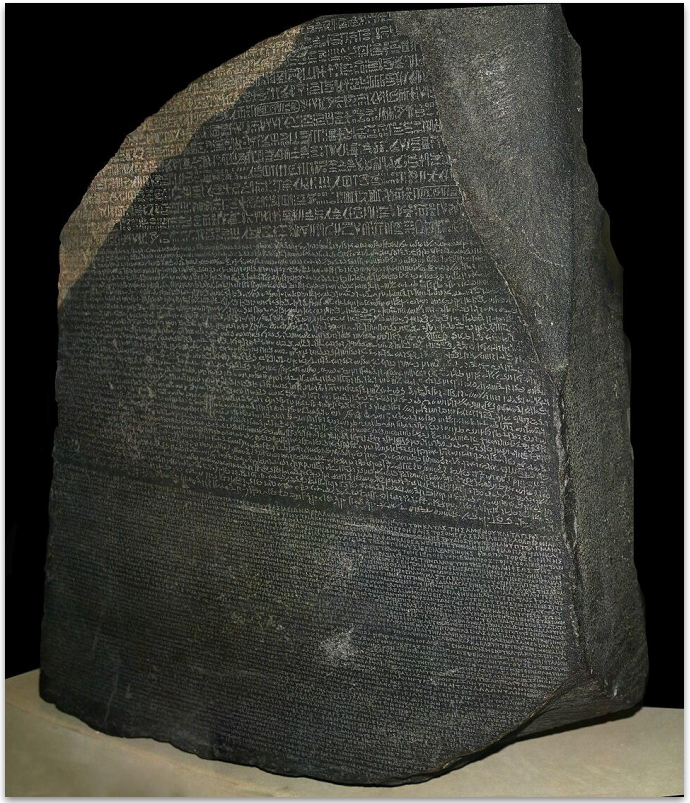
→

```
struct Muon {
  float px, py, pz;
};
class MyClass {
  std::vector<Muon> muons;
};
```

- ❌ Avoid types that are require experiment-specific libraries to make sense of the data

“Rosetta stone”

- Many analyses use **TTree::Draw,** which isn't available for RNTuple
- Moving to RDataFrame has benefits:
  - TTree *or* RNTuple inputs
  - Multithreading / distributed execution
  - Many histograms in one run
  - Easy to extend to more complicated workflows
- "Rosetta Stone"
  - TTree::Draw to RDF translation table
  - Seamless transition to RNTuple



Translating TTree::Draw to RDataFrame

👉 Rosetta Stone

TTree::Draw	ROOT::RDataFrame

`// Get the tree and Draw a histogram of x for selected y values`
`auto *tree = file->Get<TTree>("myTree");`
`tree->Draw("x", "y > 2");`
	`ROOT::RDataFrame df("myTree", file);`
	`df.Filter("y > 2").Histo1D("x")->Draw();`

`// Draw a histogram of "jet_eta" with the desired weight` `tree->Draw("jet_eta", "weight*(event == 1)");`	`df.Filter("event == 1").Histo1D("jet_eta", "weight")->Draw();`
`// Draw a histogram filled with values resulting from calling a method of the class of the `event` branch in the TTree.` `tree->Draw("event.GetNtrack()");`	`df.Define("NTrack","event.GetNtrack()").Histo1D("NTrack")->Draw();`
`// Draw only every 10th event` `tree->Draw("fNtrack","fEvtHdr.fEvtNum%10 == 0");`	`// Use the Filter operation together with the special RDF column: `rdfentry_`` `df.Filter("rdfentry_ % 10 == 0").Histo1D("fNtrack")->Draw();`
`// object selection: for each event, fill histogram with array of selected pts` `tree->Draw('Muon_pt', 'Muon_pt > 100');`	`// with RDF, arrays are read as ROOT::VecOps::RVec objects` `df.Define("good_pt", "Muon_pt[Muon_pt > 100]").Histo1D("good_pt")->Draw();`

`// Draw the histogram and fill hnew with it`
`tree->Draw("sqrt(x)>>hnew","y>0");`
 	
`// Retrieve hnew from the current directory`
`auto hnew = gDirectory->Get<TH1F>("hnew");`
	`// We pass histogram constructor arguments to the Histo1D operation, to easily give the histogram a name`
	`auto hist = df.Define("sqrt_x", "sqrt(x)").Filter("y>0").Histo1D({"hnew","hnew", 10, 0, 10}, "sqrt_x");`

TTree::Draw

```
// Fill sqrt(x) into a histogram called hnew, filter based on y
tree->Draw("sqrt(x)>>hnew(10,0,10)", "y>0");

// Retrieve hnew from the current directory
auto hnew = gDirectory->Get<TH1F>("hnew");
```

ROOT::RDataFrame

```
// Compose Define, Filter and Histo operations
auto hist = df.Define("sqrt_x","sqrt(x)")
              .Filter("y>0")
              .Histo1D({"hnew","hnew", 10, 0, 10}, "sqrt_x");
```

RNTupleProcessor

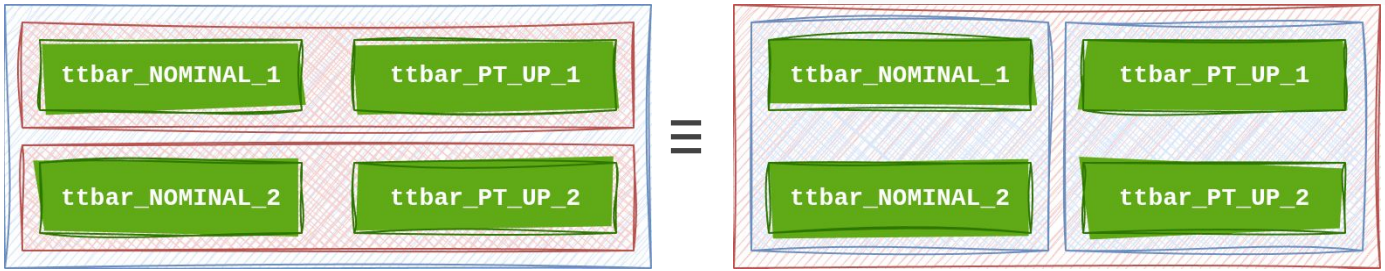
⚠️ *Stabilization is planned for ROOT 6.42 (Nov 2026)*

- Specification and data processing interface for **combined data sets**
- join and chain operations can be arbitrarily composed, always with the **same processing interface**
- Internal handling of missing field values for certain entries

- Currently available in ROOT::Experimental
  - Interface foreseen to be stabilized with ROOT v6.42 (Nov. 2026)
- Will become RNTuple-based backend to RDataFrame
  - Including a robust, non-expert-friendly way to specify dataset combinations (*RDatasetSpec v2.0*)

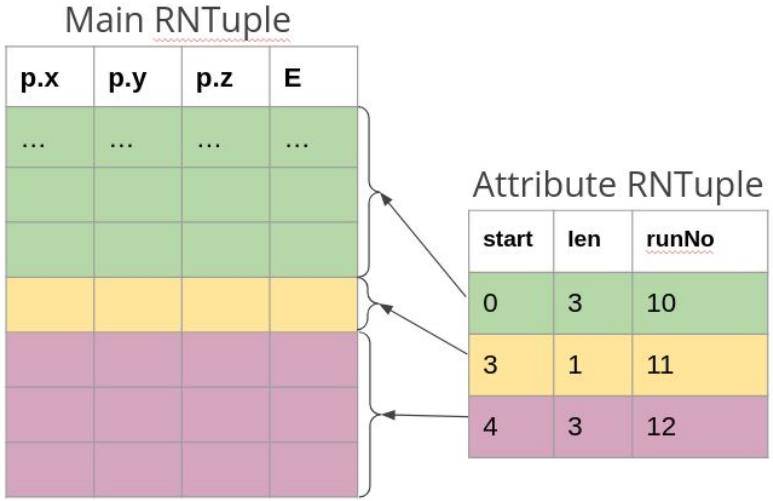
```
// Specify combination
auto proc = RNTupleProcessor::CreateJoin(
  {"nominal", "data1.root"},
  {"sys_up", "data_SYS1.root"},
  {"event_idx"}
);
// Request fields for reading
auto pt_nominal = proc->RequestField<float>("pt");
auto pt_up = proc->RequestField<float>("sys_up.pt");
// Run event loop
for (auto idx : *proc) {
  if (pt_up.HasValue())
    my_func(*pt_nominal, *pt_up);
}
```

Join field



Metadata support: RNTuple Attributes

- **Attributes** provide a native way to handle metadata in RNTuple
  - provenance, condition data, process annotations, …
- Map arbitrary metadata to ranges of data in the main RNTuple
- Data-metadata relationship persistified on disk
- API similar to the “regular data” API
- Automatically mergeable



⚠ *Stabilization is planned for ROOT 6.42 (Nov 2026)*

SoA support in RNTuple

We (usually) reason about our data as arrays of structs…

```
struct Particle {
   float fPt;
   float fEnergy;
};

using ParticleAoS = ROOT::RVec<Particle>;
```

vs.

…but sometimes representing them as structs of arrays gives better performance

```
struct ParticleSoA {
   using Record_t = Particle;

   ROOT::RVec<float> fPt;
   ROOT::RVec<float> fEnergy;
};
```

- RNTuple's columnar layout already stores data "SoA-like" on disk
- If SoA is desirable in memory as well, RNTuple provides limited support today:
  ➜ reading SoA data via RNTuple views
- We will expand SoA support both for reading and writing
  (item in this year's Plan of Work)

⚠️ *Stabilization is planned for ROOT 6.42 (Nov 2026)*

… and more

New Histogram Package

- Modern histogram package for ROOT 7
  - Light on templates (only bin content type)
  - Arbitrary dimensions with dynamic axis types at run-time
  - Simpler and sound interfaces
  - Efficient concurrent filling (memory and performance)
- First functionality available in ROOT 6.38, significantly expanded in ROOT 6.40
  - Atomic filling
  - Conversion to TH1
  - Integration into RDataFrame

👉ROOT's New Histograms

```
using namespace ROOT::Experimental;
RRegularAxis axis(40, {0.0, 20.0});

RHist<int> hist(axis, axis);
hist.Fill(8.5, 9.5);
```

Any dimension possible

ROOT 6.40

see also the tutorials

ROOT: More and more a Python Package

ROOT has a Python Interface, becoming more and more **Pythonic**

ROOT is **interoperable** with several Python packages (e.g., UHI)

ROOT is installable with `pip` (Linux, new in 2025)

ROOT is installable with **Conda** (since years)

ROOT is taught with Python **Notebooks**

ROOT is part of the open source Python ecosystem

There is no “ROOT-Python ecosystem dichotomy”

The ROOT installation is small. A Comparison of Python virtual environment sizes: `pip install` … `root`: 649 MB 💿 … `torch scipy`: 7.1 GB … `jax numba scipy`: 785 MB

One of the many Examples of Interoperability

```
$ pip install root mplhep hist numba cffi
```

Install ROOT and other packages with pip

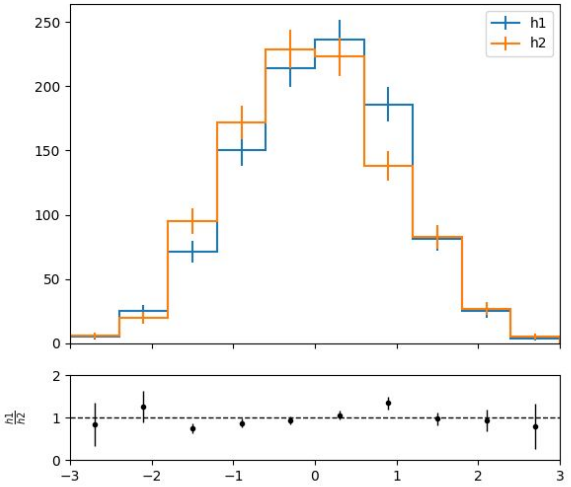
```
import numpy
import ROOT
import hist
import mplhep as hep

h1 = (
    ROOT.RDataFrame(1000)
        .Define("gauss", lambda : numpy.random.normal())
        .Histo1D(("h1", "h1", 10, -3, 3), "gauss")
        .GetValue()
)

h2 = hist.Hist(hist.axis.Regular(10, -3, 3))
h2[...] = [6, 20, 95, 172, 229, 223, 138, 83, 27, 5]

fig, _, _ = hep.comp.hists(h1, h2)
fig.savefig("plot.png")
```

Mimic an analysis with RDataFrame, using Python callables to do calculations and obtain a ROOT histogram

Create a Boost histogram

Plot the ratio of the Boost and ROOT histo with MatplotLib

The result!



Possible also thanks to ROOT histograms honouring the Uniform Histogram Interface since 6.36 (May 2025), for ratio plots in mplhep version 1.0. Work by S. Taider, co-author of UHI

Object Auto Registration: Opt Out

- Many objects register themselves to ROOT directories
  - Can lead to surprises for Histograms, TGraph2D, TEfficiency, …
- In ROOT 7, most implicit registration will be off
  - With possibility to opt in
  - And targeted exceptions (TTree::Draw, …)
- ROOT 6.40 has an opt-out mode to test this feature

```
root [0] auto dir = std::make_unique<TDirectory>("dir", "dir"); dir->cd();
root [1] auto histo = std::make_unique<TH1D>("histo", "histo", 10, 0, 10);
root [2] dir.reset();
root [3] histo->Print();
 *** Break *** segmentation violation
```

Object Auto Registration: Opt Out

- Many objects register themselves to ROOT directories
  - Can lead to surprises for Histograms, TGraph2D, TEfficiency, …
- In ROOT 7, most implicit registration will be off
  - With possibility to opt in
  - And targeted exceptions (TTree::Draw, …)
- ROOT 6.40 has an opt-out mode to test this feature

```
root [0] ROOT::Experimental::DisableObjectAutoRegistration();
root [1] auto dir = std::make_unique<TDirectory>("dir", "dir"); dir->cd();
root [2] auto histo = std::make_unique<TH1D>("histo", "histo", 10, 0, 10);
root [3] dir.reset();
root [4] histo->Print();
TH1.Print Name  = histo, Entries= 0, Total sum= 0
```

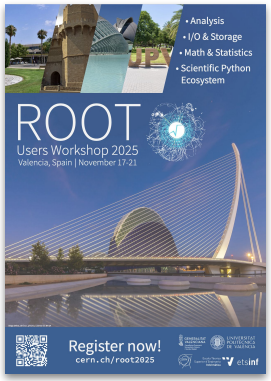
ROOT 6.40

Summary

Hackathons, Trainings and Workshops

Interactions with its community are vital for the ROOT Project, and they happen through many channels

- The ROOT Users Forum and Mattermost team
- Conferences, scientific events and publications
- ROOT Open Projects
- ROOT-organised hackathons and workshops
- Experiment's hackathons, meetings and events (not only LHC)
  - 2025: *2.5 FTE-months invested, with a huge return*
  - Thanks for inviting us!
  - Similar commitment for 2026
- Training
  - 2025: 8 trainings, >400 attendees
  - Diversifying the offer: advanced course, concept for a C++ course…
- Discussions, emails, ad-hoc meetings…



Never hesitate to reach out: we are interested in your input!

👉 Teaching ROOT

Conclusions

- **ROOT evolves, and results are visible already today**
- If you can **move to modern ROOT interfaces now**, the jump height will be minimal
  - Benefit from more performance and/or modernised interfaces already now
  - Modern code encouraged to use R... classes
- Careful deprecation strategy
  - Most T... classes (e.g., TTree, TH1) are there to stay
  - Removals deprecated for >= 1 LTS release
- ROOT has **solid yearly plans and a clear strategy for the 5 years horizon**
  - Innovation, sustainability and HL-LHC resource challenge
  - Usability, Python-C++ interfaces and obtainability
- **Sustained by community effort, development is high-paced**
  - The project keeps up with opened tickets, and then some

Thank you