Современные пакеты обработки данных

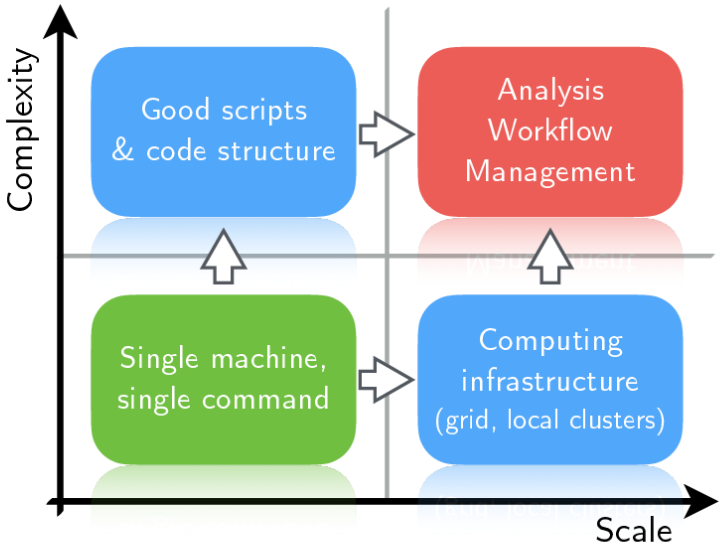
II выездное собрание "Дискуссионного клуба"

27 августа 2026 г.

Оглавление

- Most analyses are both **large and complex**
  - Structure & requirements between workloads mostly undocumented
  - Manual execution & steering of jobs, bookkeeping of data across storage elements, different data revisions, ...

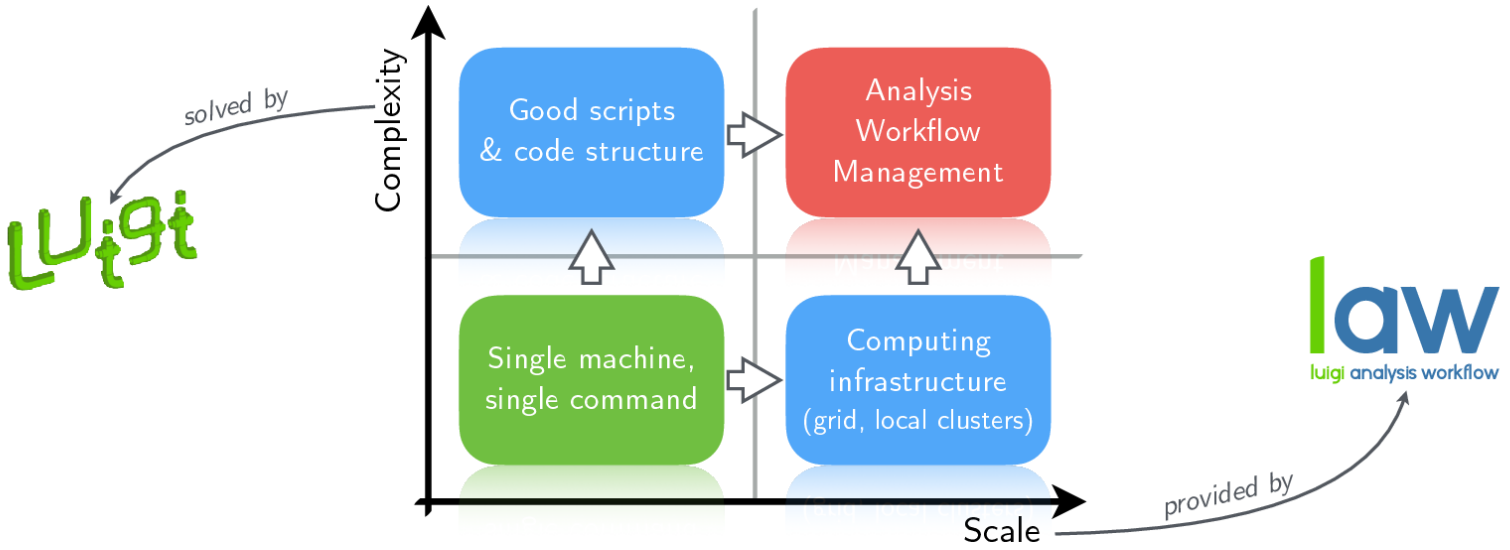
  → **Time-consuming** & **error-prone**



- **Workflow management must ...**
  - **provide full automation** → Execution through **a single command**
  - **cover all possible use cases** → Examples on next slides

- Most analyses are both **large and complex**
  - Structure & requirements between workloads mostly undocumented
  - Manual execution & steering of jobs, bookkeeping of data across storage elements, different data revisions, ...

  → **Time-consuming** & **error-prone**



- **Workflow management must ...**
  - **provide full automation** → Execution through **a single command**
  - **cover all possible use cases** → Examples on next slides

- Python package for building complex pipelines
- Development started at Spotify, now open-source and community-driven

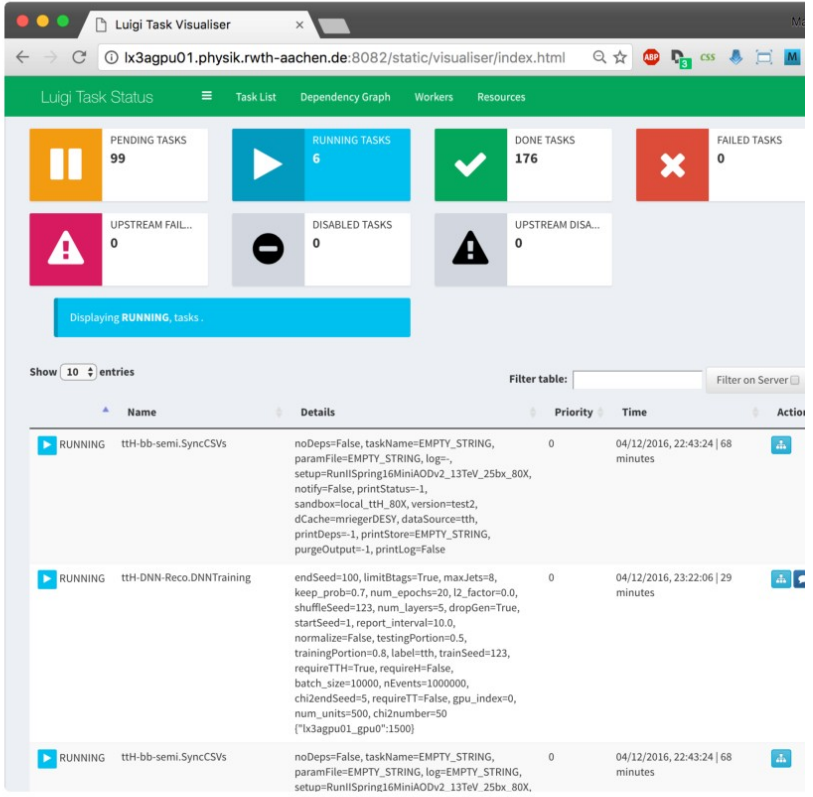
Building blocks

1. Workloads defined as **Task** classes that can **require** other **Tasks**
2. Tasks produce output **Targets**
3. **Parameters** customize tasks & control runtime behavior

- Web UI with two-way messaging (task → UI, UI → task), automatic error handling, task history browser, collaborative features, command line interface, ...
- Great documentation 📖

github.com/spotify/luigi

Watch 493 Unstar 15.2k Fork 2.3k



```python
# reco.py

import luigi

from my_analysis.tasks import Selection


class Reconstruction(luigi.Task):

    dataset = luigi.Parameter(default="ttH")


    def requires(self):
        return Selection(dataset=self.dataset)

    def output(self):
        return luigi.LocalTarget(f"reco_{self.dataset}.root")

    def run(self):
        inp = self.input()  # output() of requirements
        outp = self.output()

        # perform reco on file described by "inp" and produce "outp"
        ...
```

`Parameter` object on class-level, translates to argument parser

`string` on instance-level

luigi's local file target:
- `path: string`
- `exists(): bool`
- `remove()`
- `open(): fd`
- `...`

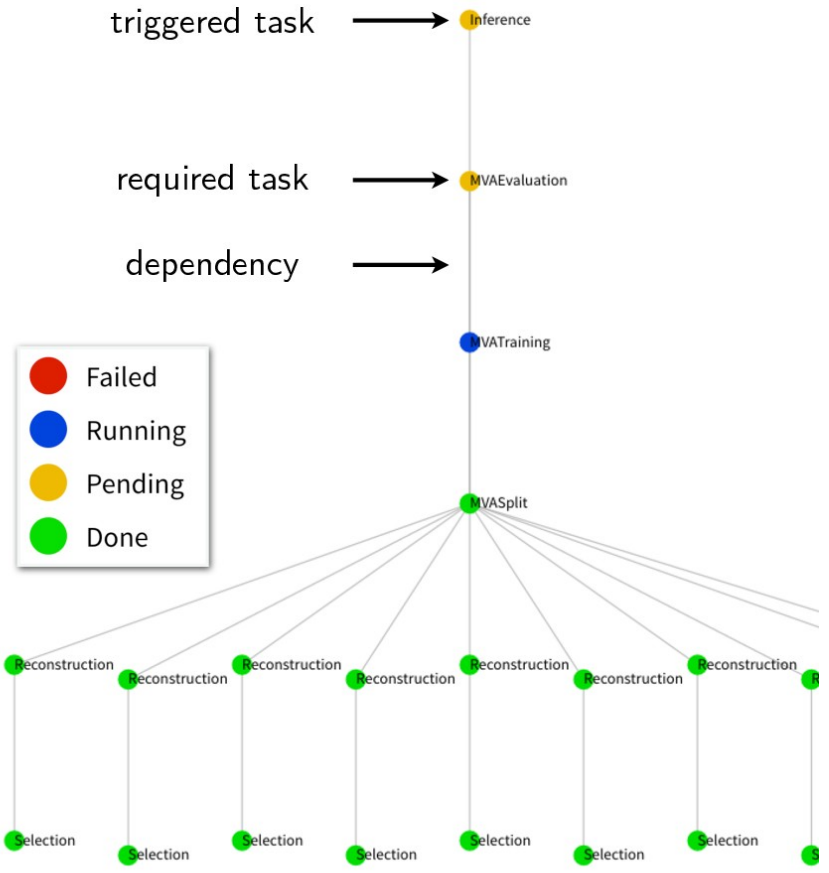
Encoding parameters into output target path

```
> python reco.py Reconstruction --dataset ttbar
```

- Luigi's execution model is make-like

  1. Create dependency tree for triggered task
  2. Determine tasks to actually run:
     - Walk through tree (top-down)
     - For each path, stop if all output targets of a task exist*

- Only processes what is really necessary
- Scalable through simple structure
- Error handling & automatic re-scheduling



* in this case, the task is considered `complete`

- **Remote storage is mandatory**
  - Local storage (e.g. at lab or institute) not always sufficient
  - Using only local storage constraints you to use only (the only?) local batch system
  - When collaborating with groups, copying files manually between sites is **error prone** & **high-maintenance**!

- **Analyses are large**
  - Imagine $\mathcal{O}(1\text{M})$ tasks $\approx \mathcal{O}(1\text{M})$ (file based) outputs and a target-based workflow engine
  - Starting the workflow requires checking the existence of many (remote) files
  - Without doing optimizations, **this will just not work** (and site admins *will* find you 👀)

- **Our IT infrastructure is (very) heterogeneous**
  - Different systems (storage, batch) and exerpise at different sites
  - Random yet typical example
    - ▻ Accessing files on site *X* via `webdav://`, and on *Y* via `root://`
    - ▻ Site *X* updates their configuration, and now `mkdir_rec` requests are no longer supported
      - Switch to `root://` on site *X* for `mkdir_rec`
    - ▻ Site *Y* updates their caching database to accelerate `stat` requests through `root://`, and now `mtime`'s are gone
      - Your local cache just got invalidated ...
      - Switch to `gsiftp://` on site *Y* for `stat`

- law: extension **on top** of *luigi* (i.e. it does not replace *luigi*)

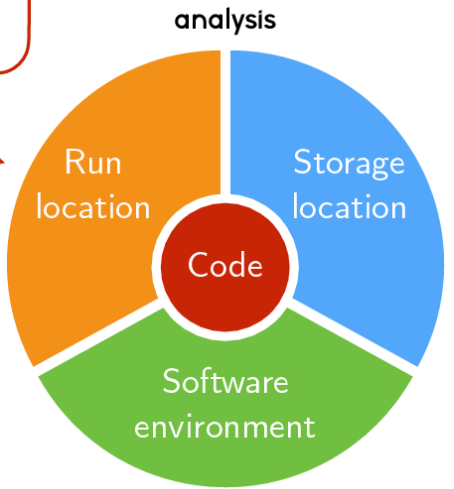
- **Design follows three primary goals**

  1. Experiment-agnostic core (in fact, not even related to physics)
  2. Scalability on HEP infrastructure (but not limited to it*)
  3. Decoupling of **run locations**, **storage locations** & **software environments**
     - ▻ Not constrained to specific resources, all components interchangeable

- Toolbox to follow an **analysis design pattern**
  - → Not a *framework* (no language or data format constraints)
  - → Serves as a **day-to-day working environment** allowing to prototype and automatically scale-out for free

- **Most used** workflow system for analyses in CMS
  - O(30) analyses, O(100) people
  - Central groups, e.g. HIG, TAU, BTV

- Also used outside CMS (e.g. LIGO) and outside HEP



- Resource-agnostic workflow management **essential** for large & complex analyses
  → Need for a flexible **design pattern** to automate arbitrary workloads

→ **End-to-end automation** of analyses over distributed resources
→ Full decoupling of **run locations**, **storage locations** & **software environments**
→ Allows to build frameworks that check every point in the CMS analysis wishlist
→ Currently working on full documentation and type annotations for next release

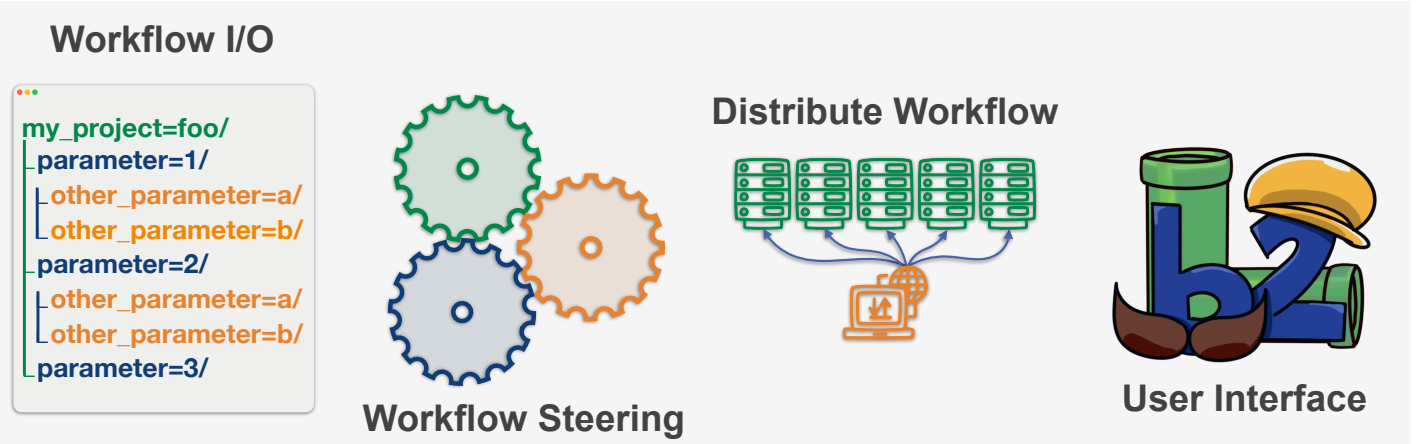
→ github.com/riga/law, law.readthedocs.io
→ github.com/spotify/luigi, luigi.readthedocs.io

Collaboration & contributions welcome!

Workflow management systems (WMS) impact

- WMS can be used as solid building blocks for community tools, adjusted to the specific workflow

Generic WMS
produced by Spotify



- Luigi as a building block for a variety of use cases:
  - b2luigi: supported by BelleII collaboration, but also used outside
  - FLARE: targeting FCC-ee analysis, well integrated with key4hep and FCCAnalysis tools
  - AWE: express ATLAS analysis in a modular fashion
  - MC-Run: end-to-end MC and analysis workflows, for precision phenomenology and beyond

ROOT 7:
Getting Ready for HL-LHC

Stephan Hageboeck [1], Jonas Hahnfeld [1,2],
Giacomo Parolini [1], Philippe Canal [3] for the ROOT Project

CHEP 2026, Bangkok
28.05.2026

[1] CERN, EP-SFT
[2] Goethe University Frankfurt
[3] FNAL, SCD

Where is ROOT headed?

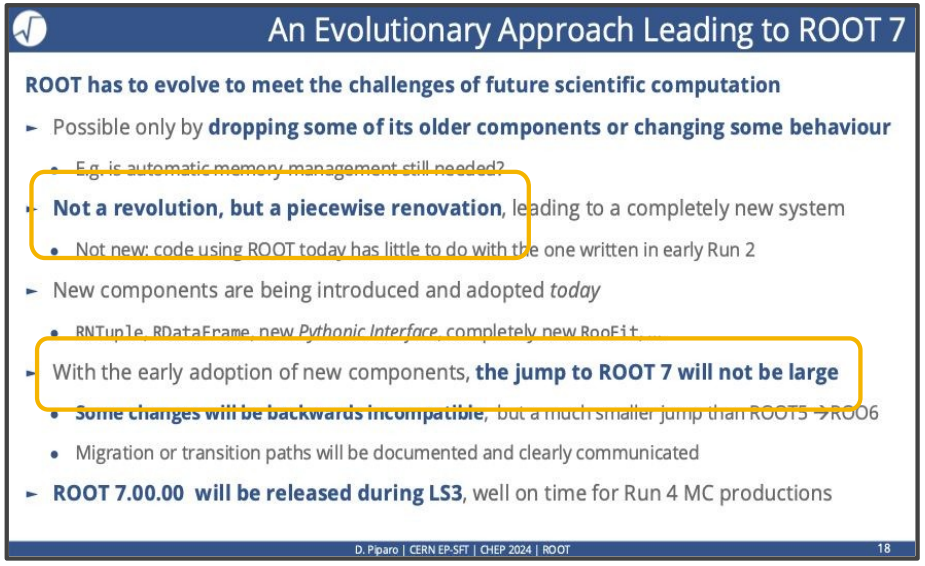
ROOT continuously evolves

- Modern interfaces
- Profit from today's hardware
- Reduce maintenance cost

ROOT will **continue to power** HEP computing

- ROOT 7 *can't be a revolution* that breaks all existing codes
- So it must be an **evolution** where **old interfaces remain available**, while **new ones are added**

D. Piparo - CHEP 2024



Evolving ROOT 6 towards ROOT 7

The evolution of ROOT is already in progress! E.g.

- RDataFrame: Multithreaded/distributed declarative analysis
- RNTuple: Fast and efficient revolution of HEP data storage
- RooFit: Same frontend, different backends, auto-diff in C++
- ROOT's Python bindings are evolving continuously

Features being added currently:

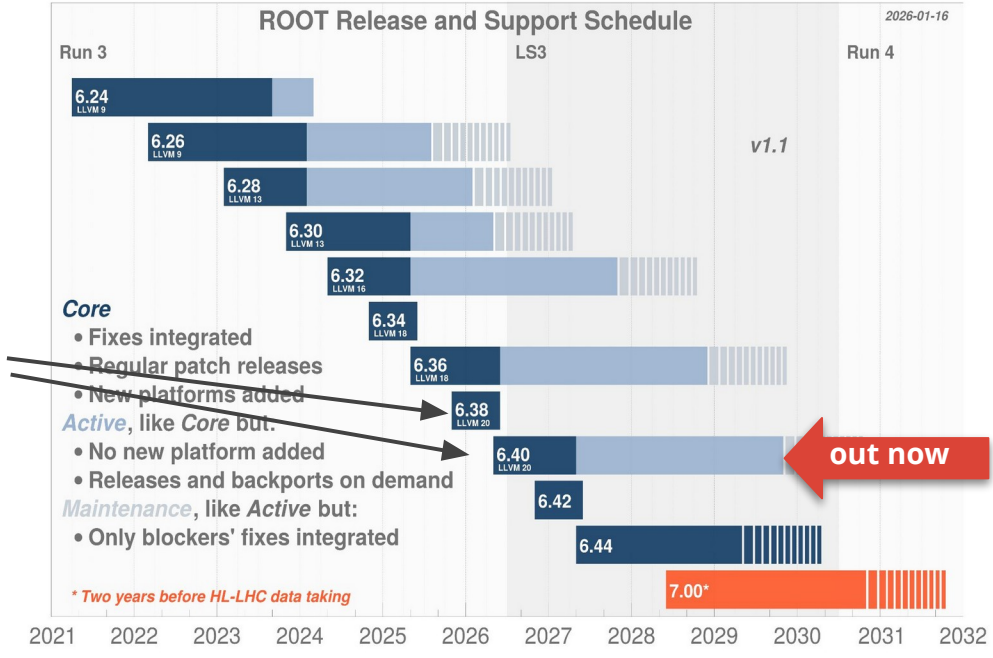
- RFile: a minimal interface to ROOT files; with clear ownership
- RHist: A new histogram package
- And a change in ROOT's object ownership model (this talk)

ROOT Release Schedule

ROOT 7 will be released two years before HL-LHC

RFile, histograms and ownership are taking shape:

- Releasing demo versions **now** (in `ROOT::Experimental`)
- In v6.42: continued development; user feedback
- Mid 2027, v6.44: Features move out of `ROOT::Experimental`



https://root.cern/install/all_releases/

A Look at Upcoming Features

RFile

A minimal, modern interface to ROOT files

With clear ownership in C++

Easy to use from Python

For a much deeper look

👉A modernized interface to ROOT files

```cpp
auto file = RFile::Recreate("/path/to/file.root");
file->Put("directory/hist", *existingHist);

std::unique_ptr<TH1D> hist =
  file->Get<TH1D>("otherDirectory/hist2");
file.reset();

hist->Draw(); // hist survives!
```

Smart pointers, clear ownership

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

ROOT 6.40

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});
```

Any dimension possible

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

ROOT 6.40

see also the tutorials

Object Auto Registration

Implicit Object Ownership

ROOT(7) – What else important ?

Looking forward to and interested in (just naming a few)

- **Modernization & performance devs**
  - RNTuple, RDataFrame, RHist, RFile, …, RGeo??
  - modern C++ with explicit ownership
  - high performance
- **Stability and robustness increases**
  - robust APIs designed to prevent bad code
  - enhanced multi-threaded safety
- **Portability considerations and improvements**
  - GPU support of basic types/algorithms
  - ARM
- **RDataFrame use expansion, …**

```
TFile *f = new TFile("data.root", "RECREATE");
TH1F *h = new TH1F("h", "", 100, 0, 10);
f->Write();
f->Close();
delete h; // segfaults
```

on Monday

S. Wenzel

on behalf of the ALICE experiment

ROOT Users Workshop 2025

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

Object Auto Registration: Details

Is this the same as TH1::AddDirectory(false)? → **No**

DisableObjectAutoRegistration() is **thread-local**

Can be used safely in multi-threaded frameworks

One can permanently change the default:

- Environment variable:
  `ROOT_OBJECT_AUTO_REGISTRATION=0 <program>`
- Entry in rootrc:
  `Root.ObjectAutoRegistration: 0`

The opposite is EnableObjectAutoRegistration()

- If needed, this can enable ROOT 6 behaviour in ROOT 7

Status of Auto-Registration Off

Auto-registration off unifies several flags & adds new opportunities

	Does it honour DisableObjectAutoRegistration() ?	Could this be disabled previously?
TH1 and derived	Yes	TH1::AddDirectoryStatus() *
TGraph2D	Yes	TH1::AddDirectoryStatus() *
RooPlot	Yes	RooPlot::addDirectoryStatus() *
TEfficiency	Yes	Inconsistent
TProfile2D	Yes	TH1::AddDirectoryStatus() *
TEntryList	Planned for 6.42	No
TEventList	Planned for 6.42	No
TFunction	Planned for 6.42 / 6.44	Only on a per-constructor basis

* Using this isn't thread safe!

Object Auto Registration: Code changes

Using auto registration off will require targeted code changes on the user side

- Either write objects explicitly
- Or use
  `obj->SetDirectory(object);`
  for deferred writing

No longer requires `outputFile->cd();`

👉A modernized interface to ROOT files

Example how the tutorials of TUnfold were prepared for auto-registration off

```
@@ -104,8 +104,6 @@ void testUnfold5c()
  TUnfoldBinning *detectorBinning,*generatorBinning;

-  outputFile->cd();
-
  // read binning schemes in XML format
 #ifndef READ_BINNING_CINT
  TDOMParser parser;
@@ -122,8 +120,8 @@ void testUnfold5c()
-  detectorBinning->Write();
-  generatorBinning->Write();
+  outputFile->WriteTObject(detectorBinning);
+  outputFile->WriteTObject(generatorBinning);
```

Or use RFile

Object Auto Registration

Using auto registration off will require targeted code changes on the user side

- Either write objects explicitly
- Or use
  `obj->SetDirectory(object);`
  for deferred writing

No longer requires `outputFile->cd();`

👉A modernized interface to ROOT files

Example how the tutorials of TUnfold were prepared for auto-registration off

```
@@ -154,10 +152,10 @@ void testUnfold5c()
  Float_t etaRec,ptRec,discr,etaGen,ptGen;
  Int_t istriggered,issignal;

-  outputFile->cd();
-
  TH1
*histDataReco=detectorBinning->CreateHistogram("histDataReco");
  TH1
*histDataTruth=generatorBinning->CreateHistogram("histDataTruth");
+  histDataReco->SetDirectory(outputFile);
+  histDataTruth->SetDirectory(outputFile);
```

Or use RFile

Testing Auto-Registration Off

ROOT 6.40 has been prepared to work both with auto-registration on and off

- 27 pull requests, 73 commits, +853 -1222 lines in 123 files

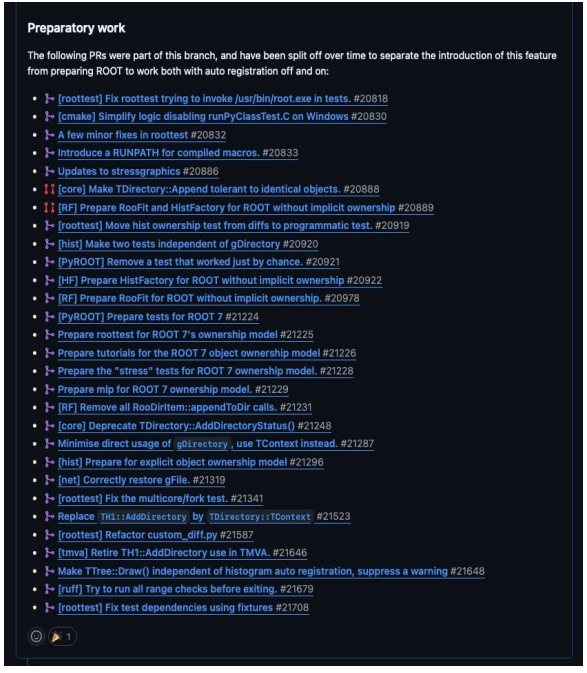
ROOT's CI includes a build with auto-registration off

- All tests/tutorials are green now

User feedback welcome!

- Test auto-registration off with your code if you can
- If something needs changes in ROOT, get in touch https://github.com/root-project/root

https://github.com/root-project/root/pull/18083



Preparatory work

The following PRs were part of this branch, and have been split off over time to separate the introduction of this feature from preparing ROOT to work both with auto registration off and on:

- [roottest] Fix roottest trying to invoke /usr/bin/root.exe in tests. #20818
- [cmake] Simplify logic disabling runPyClassTest.C on Windows #20830
- A few minor fixes in roottest #20832
- Introduce a RUNPATH for compiled macros. #20833
- Updates to stressgraphics #20886
- [core] Make TDirectory::Append tolerant to identical objects. #20888
- [RF] Prepare RooFit and HistFactory for ROOT without implicit ownership #20889
- [roottest] Move hist ownership test from diffs to programmatic test. #20919
- [hist] Make two tests independent of gDirectory #20920
- [PyROOT] Remove a test that worked just by chance. #20921
- [HF] Prepare HistFactory for ROOT without implicit ownership #20922
- [RF] Prepare RooFit for ROOT without implicit ownership. #20978
- [PyROOT] Prepare tests for ROOT 7 #21224
- Prepare roottest for ROOT 7's ownership model #21225
- Prepare tutorials for the ROOT 7 object ownership model #21226
- Prepare the "stress" tests for ROOT 7 ownership model. #21228
- Prepare mlp for ROOT 7 ownership model. #21229
- [RF] Remove all RooDirItem::appendToDir calls. #21231
- [core] Deprecate TDirectory::AddDirectoryStatus() #21248
- Minimise direct usage of `gDirectory`, use TContext instead. #21287
- [hist] Prepare for explicit object ownership model #21296
- [net] Correctly restore gFile. #21319
- [roottest] Fix the multicore/fork test. #21341
- Replace `TH1::AddDirectory` by `TDirectory::TContext` #21523
- [roottest] Refactor custom_diff.py #21587
- [tmva] Retire TH1::AddDirectory use in TMVA. #21646
- Make TTree::Draw() independent of histogram auto registration, suppress a warning #21648
- [ruff] Try to run all range checks before exiting. #21679
- [roottest] Fix test dependencies using fixtures #21708

Sustainability: Deprecations

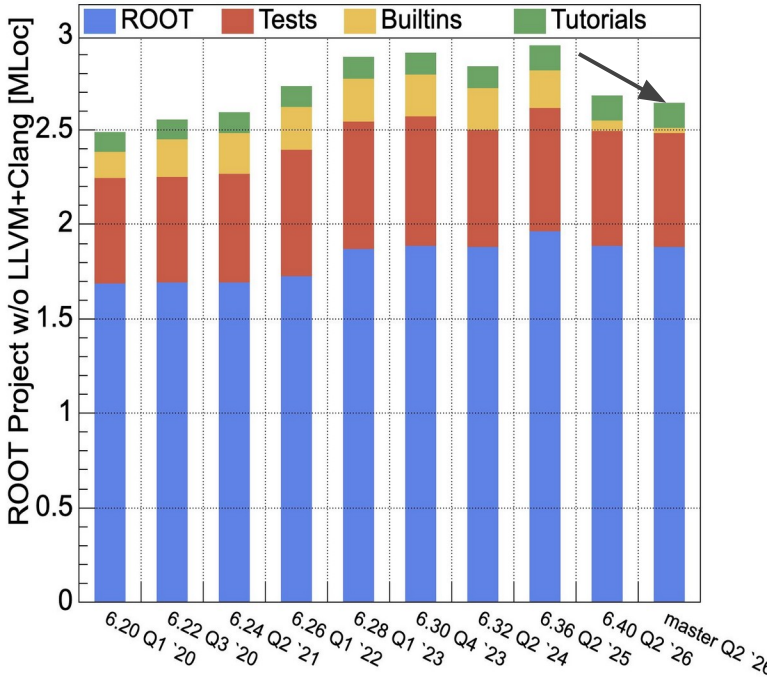
Deprecations

To keep ROOT sustainable, some components will not be carried into the future

- Already slimming down builtins
- Components that cannot be maintained
- Components superseded with better alternatives

Core ROOT functionality with a large user base **will continue to be supported**

- "Will you deprecate TTree/TFile/TH... ?"
  → No
- But no significant work will be invested if there is a better alternative



Counted with *cloc*

Deprecation Strategy

Deprecations and removals are already happening in ROOT 6

- TPythia6 was removed in v6.32*
- PROOF was removed in v6.38
- Build options are being cleaned up

Deprecations are announced in release notes & trigger warnings when building ROOT

A few versions after, such components / interfaces are removed

* TPythia6 story:

- Generators in neutrino community relied on it!
- Standalone EGPythia6 package on top of ROOT created

v6.30 Release Notes

Deprecation and Removal

- The minimum C++ standard supported by ROOT is now C++17.
- Support for Python 2 is now deprecated and it will be removed in next release 6.32.

Deprecated and removed ROOT modules

The following previously deprecated build options have been removed:

- alien
- gfal
- gsl_shared
- jemalloc
- monalisa
- pyroot_legacy
- tcmalloc
- xproofd

The following build options have now been deprecated and will be removed in the future v6.32:

- cxxmodules
- exceptions
- oracle
- pythia6
- pythia6_nolink
- pyroot-python2

Please let us know at rootdev@cern.ch if their planned removal would cause problems for you!

Summary

Summary

ROOT evolves, and results are visible already today

ROOT 7 will be released in 2028, two years before the start of HL-LHC

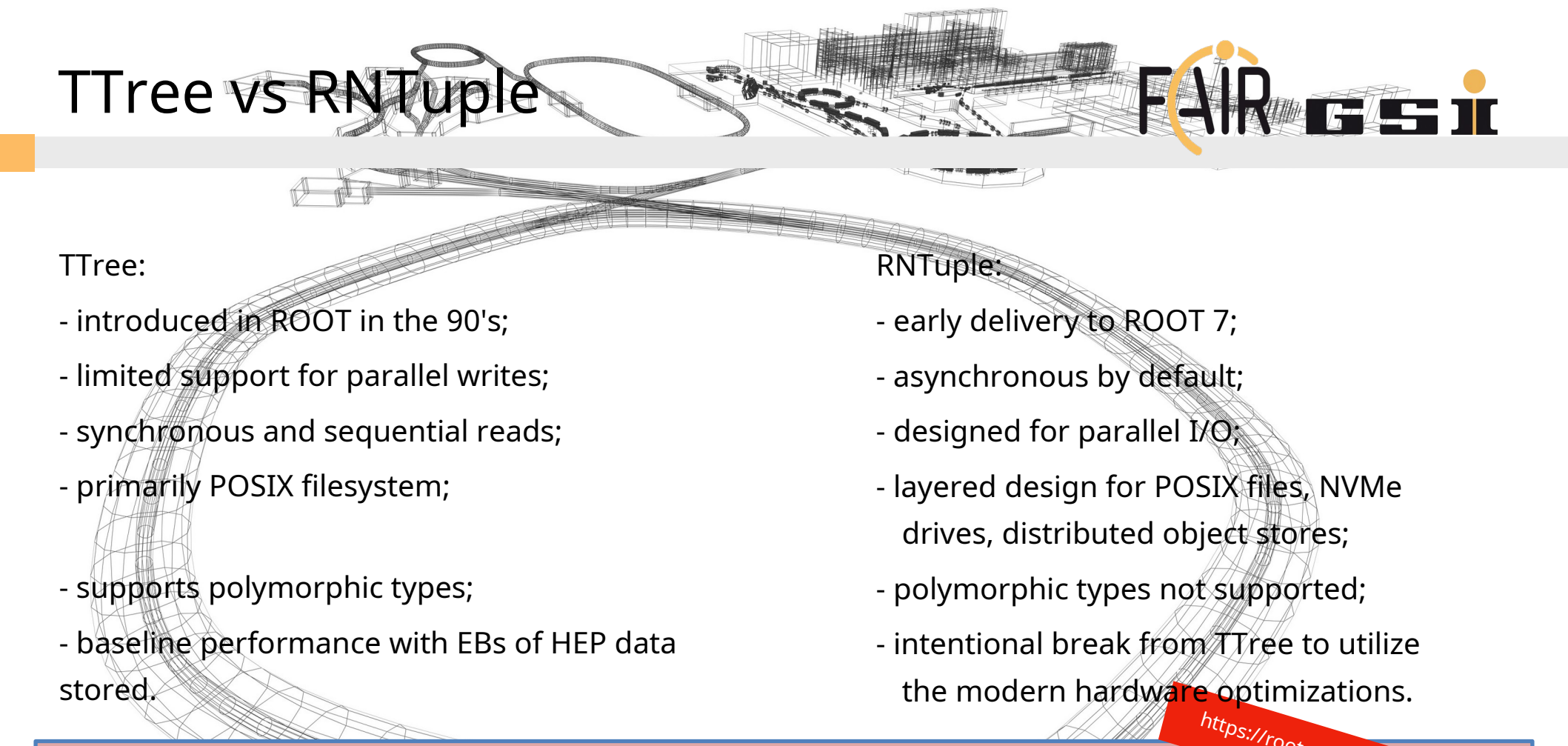
If you can move to modern ROOT interfaces now, the jump height will be minimal

- And you benefit from more performance and/or modernised interfaces already now

Start testing ROOT with auto registration off; feedback welcome

https://github.com/root-project/root

TTree vs RNTuple



TTree:

- introduced in ROOT in the 90's;
- limited support for parallel writes;
- synchronous and sequential reads;
- primarily POSIX filesystem;
- supports polymorphic types;
- baseline performance with EBs of HEP data stored.

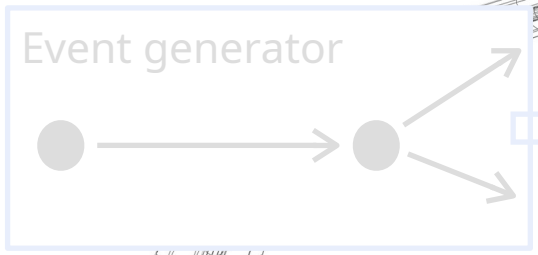
RNTuple:

- early delivery to ROOT 7;
- asynchronous by default;
- designed for parallel I/O;
- layered design for POSIX files, NVMe drives, distributed object stores;
- polymorphic types not supported;
- intentional break from TTree to utilize the modern hardware optimizations.

https://root.cern/blog/rntuple-update/

With RNTuple, we could potentially save 20-35% of storage space.

Test reconstruction

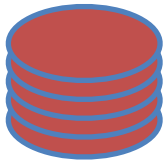


Example hit producer:

- TTree to TTree;
- RNTuple to RNTuple.

TTree

RNTuple



22% runtime
19% storage reduction

145s

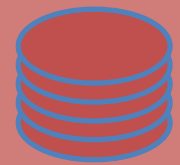
TTree

2297MB

113s

RNTuple

1874MB

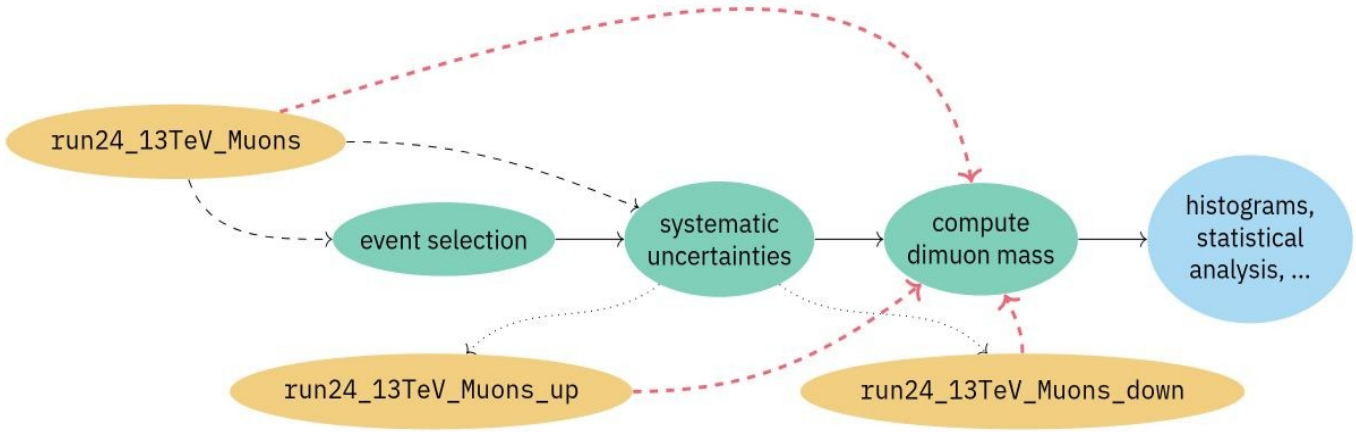


Dataset Combinations

HEP analysis workflows are increasingly growing in complexity, and often involve intermediate data products

- No straightforward way in our ecosystem to combine data from different sources
- Current practices heavily rely on data duplication

join and *union* (chain) are common operations in database systems, can we apply them to HEP?



Original work by F. de Geus

New Developments in ROOT's RDataFrame

Marta Czurylo [1], Stephan Hageböck [1], Vincenzo Eduardo Padulano [1]

CHEP 2026, Bangkok
25.05.2026

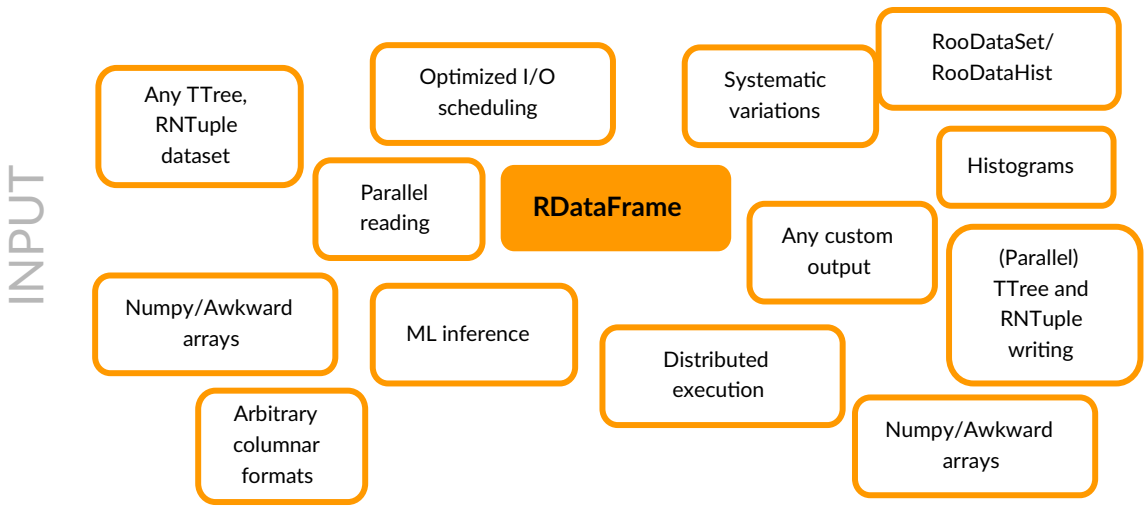
[1] CERN,

RDataFrame in a Nutshell

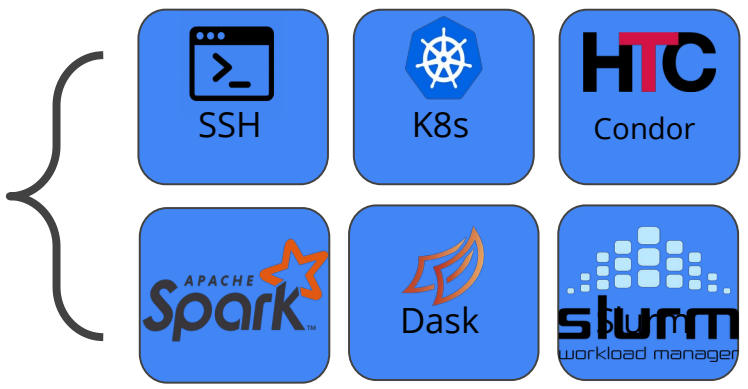

```
df = ROOT.RDataFrame(treeName/RNTupleName, fileNames)
histo = df.Define(...).Filter(...).Vary(...).Histo1D(...)
histo.Snapshot(...)
```

RDataFrame: ROOT's declarative interface for analysis...

- ROOT's high-level interface for analysis



- Supports native parallel execution
- Multi-threaded and distributed

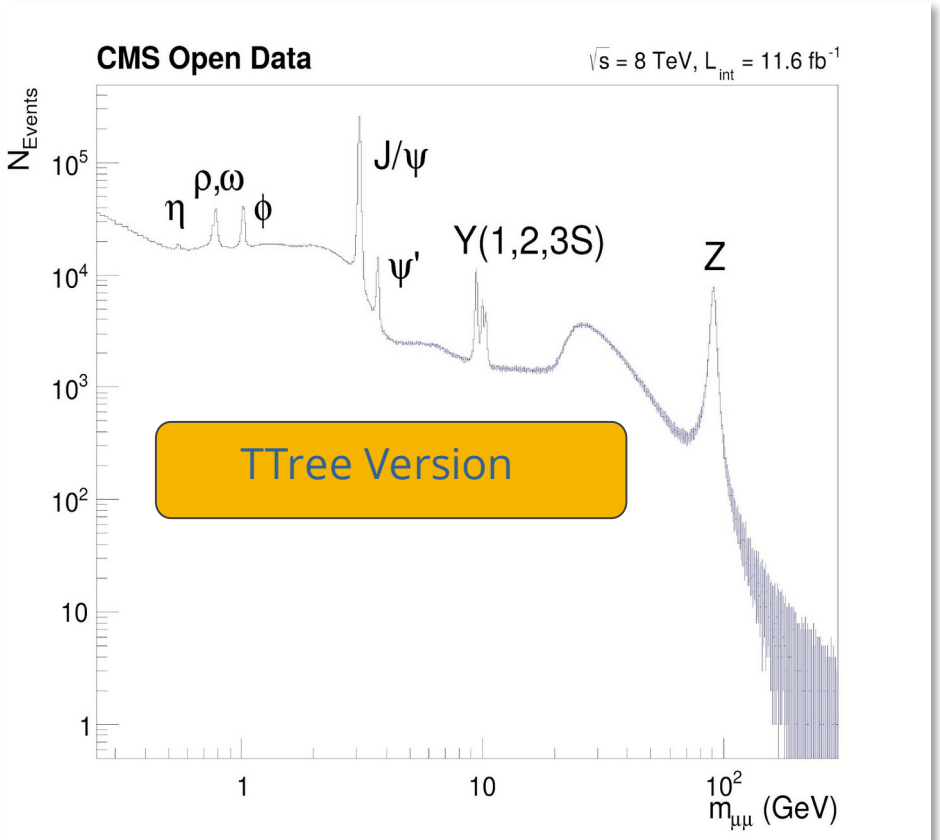


Modern Data Analysis with RNTuple?

TTree - RDataFrame Analysis

For more details see the di-muon analysis tutorial

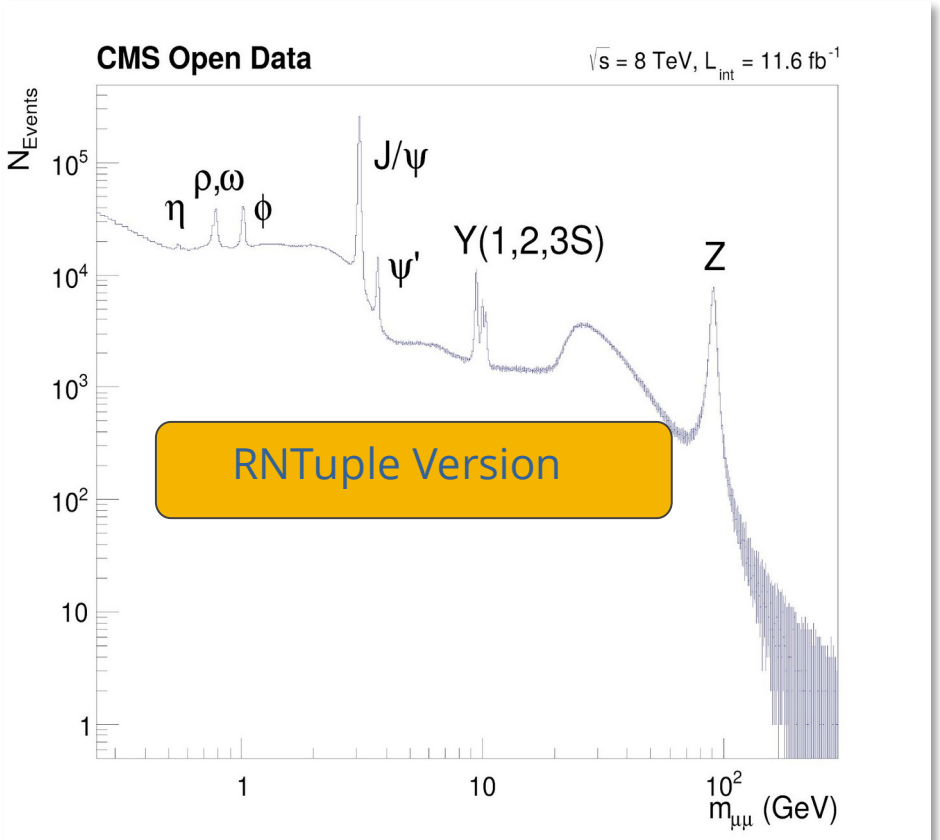
```
import ROOT

ROOT.EnableImplicitMT()

# Create dataframe from NanoAOD files
dataset = 'root://eospublic.cern.ch//eos/opendata/cms/derived-data/AOD2NanoAODOutreachTool/Run2012BC_DoubleMuParked_Muons.root'
df = ROOT.RDataFrame ('Events', dataset)

# Select only events with exactly two muons and require opposite charge
df_2mu = df.Filter('nMuon == 2',
                   'Events with exactly two muons')
df_os = df_2mu.Filter('Muon_charge[0] != Muon_charge[1]',
                      'Muons with opposite charge')

# Compute invariant mass of the dimuon system
df_mass = df_os.Define('Dimuon_mass',
                       'InvariantMass(Muon_pt, Muon_eta, Muon_phi, Muon_mass)')

# Make histogram of dimuon mass spectrum. Note how we can set titles and axis labels in one go.
h = df_mass.Histo1D(('', 'Dimuon mass;m_{#mu#mu} (GeV);N_{Events}',
                     30000, 0.25, 300), 'Dimuon_mass')
# Request cut-flow report
report = df_mass.Report()

# Produce plot
ROOT.gStyle.SetOptStat(0); ROOT.gStyle.SetTextFont(42)
c.SetLogx(); c.SetLogy()
h.GetXaxis().SetTitleSize(0.04)
h.GetYaxis().SetTitleSize(0.04)
h.Draw()
label = ROOT.TLatex(); label.SetNDC(True)
label.DrawLatex(0.175, 0.740, '#eta'); label.DrawLatex(0.205, 0.775, '#rho,#omega')
label.DrawLatex(0.270, 0.740, '#phi'); label.DrawLatex(0.400, 0.800, 'J/#psi')
label.DrawLatex(0.415, 0.670, '#psi'); label.DrawLatex(0.485, 0.700, 'Y(1,2,3S)')
label.DrawLatex(0.755, 0.680, 'Z')
label.SetTextSize(0.04); label.DrawLatex(0.10, 0.92, '#bf{CMS Open Data}')
label.SetTextSize(0.03); label.DrawLatex(0.63, 0.92, '#sqrt{s} = 8 TeV, L_{int} = 11.6 fb^{-1}')
```



RNTuple - RDataFrame Analysis

For more details see the di-muon analysis tutorial

```
import ROOT

ROOT.EnableImplicitMT()

# Create dataframe from NanoAOD files
dataset = 'root://eospublic.cern.ch//eos/root-eos/cms_opendata_2012_nanoaod/rntuple/Run2012BC_DoubleMuParked_Muons.root'
df = ROOT.RDataFrame ('Events', dataset)

# Select only events with exactly two muons and require opposite charge
df_2mu = df.Filter('nMuon == 2',
                   'Events with exactly two muons')
df_os = df_2mu.Filter('Muon_charge[0] != Muon_charge[1]',
                      'Muons with opposite charge')

# Compute invariant mass of the dimuon system
df_mass = df_os.Define('Dimuon_mass',
                       'InvariantMass(Muon_pt, Muon_eta, Muon_phi, Muon_mass)')

# Make histogram of dimuon mass spectrum. Note how we can set titles and axis labels in one go.
h = df_mass.Histo1D(('', 'Dimuon mass;m_{#mu#mu} (GeV);N_{Events}',
                     30000, 0.25, 300), 'Dimuon_mass')
# Request cut-flow report
report = df_mass.Report()

# Produce plot
ROOT.gStyle.SetOptStat(0); ROOT.gStyle.SetTextFont(42)
c.SetLogx(); c.SetLogy()
h.GetXaxis().SetTitleSize(0.04)
h.GetYaxis().SetTitleSize(0.04)
h.Draw()
label = ROOT.TLatex(); label.SetNDC(True)
label.DrawLatex(0.175, 0.740, '#eta'); label.DrawLatex(0.205, 0.775, '#rho,#omega')
label.DrawLatex(0.270, 0.740, '#phi'); label.DrawLatex(0.400, 0.800, 'J/#psi')
label.DrawLatex(0.415, 0.670, '#psi'); label.DrawLatex(0.485, 0.700, 'Y(1,2,3S)')
label.DrawLatex(0.755, 0.680, 'Z')
label.SetTextSize(0.04); label.DrawLatex(0.10, 0.92, '#bf{CMS Open Data}')
label.SetTextSize(0.03); label.DrawLatex(0.63, 0.92, '#sqrt{s} = 8 TeV, L_{int} = 11.6 fb^{-1}')
```



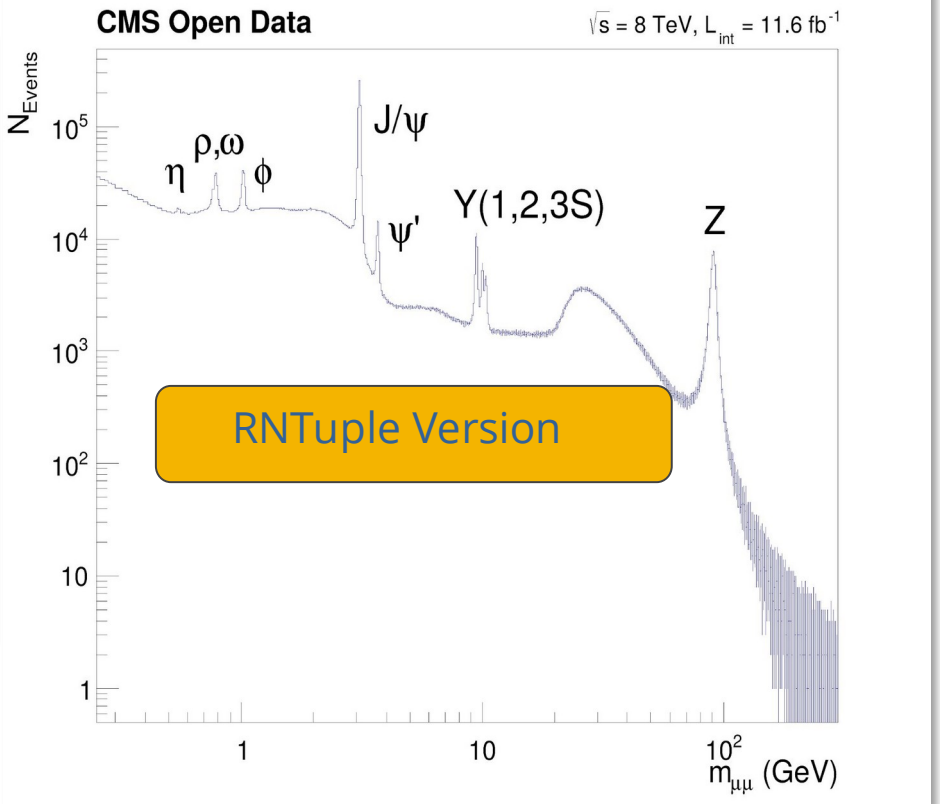
RNTuple - RDataFrame Analysis

For more details see the di-muon analysis tutorial

```
import ROOT

ROOT.EnableImplicitMT()

# Create dataframe from NanoAOD files
dataset = 'root://eospublic.cern.ch//eos/root-eos/cms_opendata_2012_nanoaod/rntuple/Run2012BC_DoubleMuParked_Muons.root'
df = ROOT.RDataFrame ('Events', dataset)

# Sele
df_2

df_o


# Co
df_m


# Ma
h =

# Re
repo

# Pr
ROOT
c.Se
h.Ge
h.Ge
h.Dr
labe
labe
labe
label.DrawLatex(0.755, 0.680, 'Z')
label.SetTextSize(0.04); label.DrawLatex(0.10, 0.92, '#bf{CMS Open Data}')
label.SetTextSize(0.03); label.DrawLatex(0.63, 0.92, '#sqrt{s} = 8 TeV, L_{int} = 11.6 fb^{-1}')
```

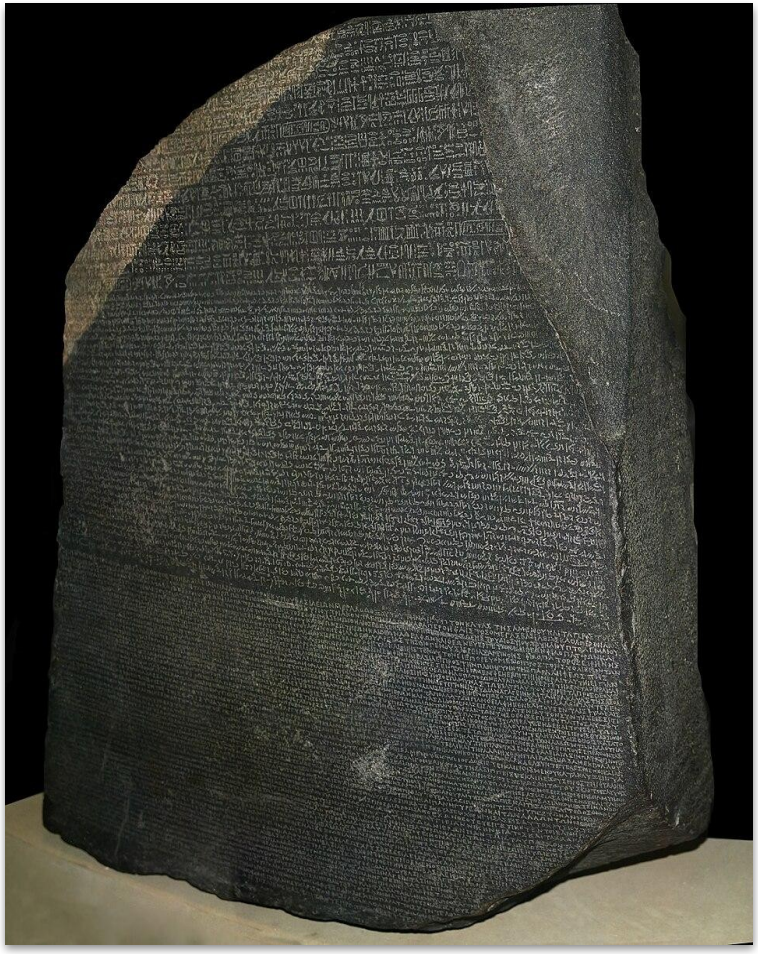
Identical code – **Better** performance

- **Smaller** input files
- **Seamless** entry point to RNTuple
- Feature parity between TTree and RNTuple processing



"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");
```

Improvements in RDataFrame:

"User Defines *What*,
RDataFrame Determines *How*"

More Speed

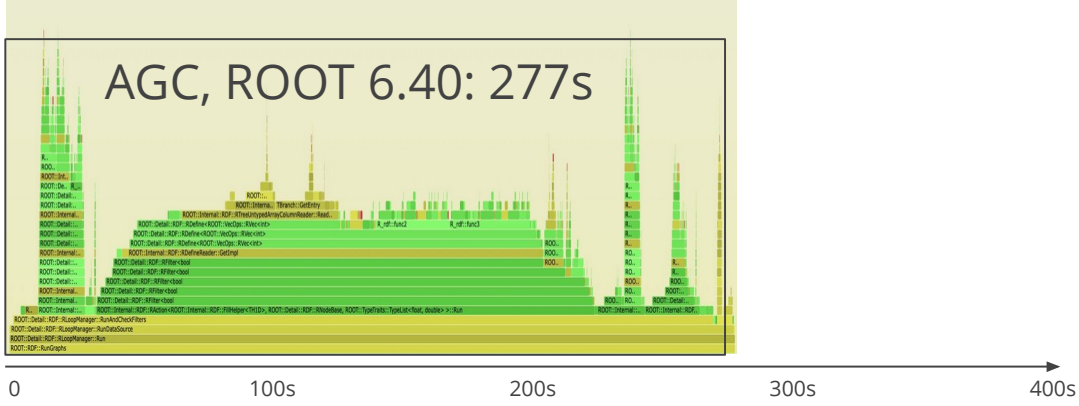
Internal refactoring of RDataFrame to unify processing of TTree and RNTuple

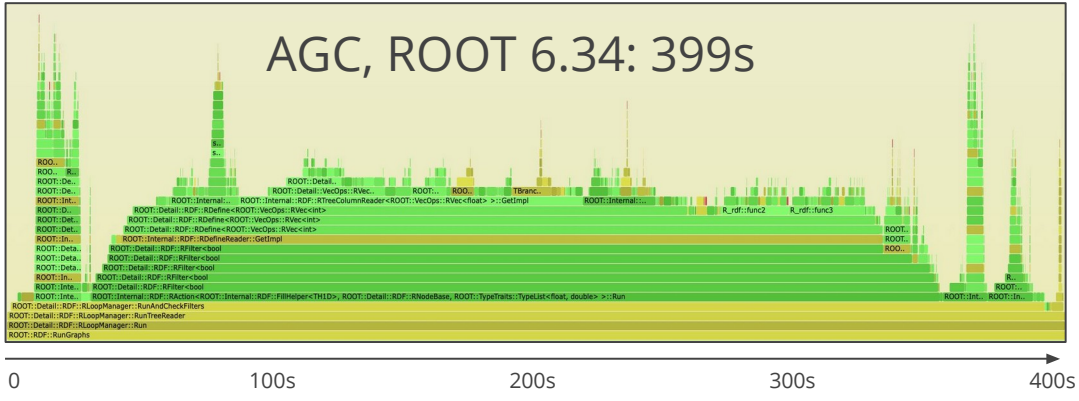
Allowed for overhauling the way to access data

No changes in User Code

The data-processing part of the Analysis Grand Challenge (AGC) is **30%** faster now

AGC, RDataFrame for reading, filtering and histogramming, local TTree dataset with zstd compression, 1 thread





Better Memory Usage

For complicated analyses, the RDataFrame computation graph might contain thousands of nodes

Developers of the FastFrames framework ran into memory pressure, got in touch with the ROOT team

A trivial RDataFrame with 20k nodes:

- CHEP24: 23.2 GB
- CHEP26: **3.6 GB (-85%)**

No code change

```
for (std::size_t i = 0; i < 20000; i++) {
    node = node.Define(
    "col_" + std::to_string(i + 1),
     [](){ return 42; }
    );
 }
```

Simplifying Systematic Variations

Systematic variations can either be compiled ahead of time (framework developers / expert users)

```
df.Vary({"pt", "eta"},  // the columns that will vary simultaneously
    [](float pt, float eta) { return RVec<RVecF>{{pt*0.9, pt*1.1}, {eta*0.9, eta*1.1}}; },
    {"pt", "eta"},  // inputs to the Vary expression
    {"down", "up"}, "ptAndEta"); // variations tags, variation name
```

Or just-in-time compiled to simplify the syntax and the usage from Python

- RDataFrame infers input colums to the vary expression
- New in ROOT 6.40: RDataFrame also infers return types

```
df.Vary({"pt", "eta"},
      R"({
       {pt *0.9, pt *1.1}, // pT variation
       {eta*0.9, eta*1.1}  // eta variation
      })",
      {"down", "up"}, "ptAndEta");
```

New Features in RDataFrame

Zero-overhead ML training with ROOT

Go from ROOT to ML training without intermediate files

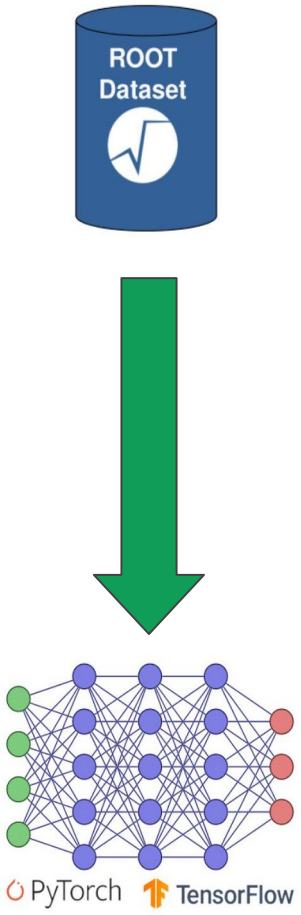
Leverage all the power of RDataFrame and ROOT's I/O

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

```
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)
```

RHist in RDataFrame

In ROOT 6.40, RDataFrame has experimental support for ROOT's new histograms

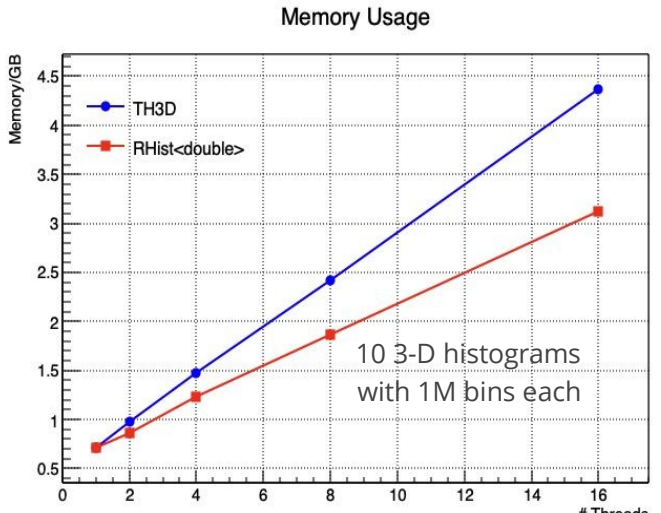
Atomic filling: no need to clone histograms per thread

Save memory when running MT with high-dimensional histograms

More technical details:

👉ROOT's New Histograms

```
using namespace ROOT::Experimental;
std::vector<RAxisVariant> axes{
  RRegularAxis{100, {0., 100.}},
  RRegularAxis{20, {-10., 10}},
  ...
};
auto h = rdf.Hist<double>(axes,
                          {"x", "y"});
```



Template-free Snapshots

- The **Snapshot** operation saves RDataFrame columns to disc
- Snapshot required specifying all column types
  - Either **explicitly** (**template parameters written by the user**)
  - Or **implicitly**, by **JIT** compiling the **template**
  - Many columns → **very long** compilation times ([1],[2],[3])
- **Now:** Snapshot **does not** need column types for compilation [4]!
  - If you wrote Snapshot<**int**, **float**, ...>, you can replace it with just Snapshot!
  - If you wrote Snapshot before, now it will be much faster!

Example: Snapshotting 500 trivial columns

- User time: 12.4 s → 4.1 s
- Memory: 1.2 G → 600 M

→

```
ROOT::RDataFrame rdf(10);
auto df = rdf.Define("x", "rdfentry_");
for (unsigned int i = 0; i < 500; ++i) {
  df = df.Define("x" + std::to_string(i), "x*x");
}
df.Snapshot("tree", "/tmp/testSnapshot.root");
```

Snapshots with Systematic Variations

Systematic variations supported in RDataFrame since ROOT 6.26

Long-standing request: saving systematic variations using Snapshot

```
auto s = 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;
```

One flag enables snapshot of variations

Experimental in ROOT v6.40 (out now)

Snapshots with Variations: Dataset on Disk

- Dataset schema preserved: **input types == output types**
- For any variation that doesn't pass its Filter(), default-constructed values get written
- When read with RDataFrame, these values will be masked

Row	R_rdf_mask_t_0	x	y	x__xVariation_0	y__xVariation_0	x__xVariation_1	y__xVariation_1
0	101b	20	-20	0	0	21	-21
1	111b	30	-30	29	-29	31	-31
2	111b	40	-40	39	-39	41	-41
3	110b	50	-50	49	-49	51	-51
4	111b	60	-60	59	-59	61	-61
5	010b	70	-70	69	-69	0	0
6	101b	20	-20	0	0	21	-21
7	111b	30	-30	29	-29	31	-31
8	111b	40	-40	39	-39	41	-41

Snapshots with Variations: Dataset on Disk

- Dataset schema preserved: **input types == output types**
- For any variation that doesn't pass its Filter(), default-constructed values get written
- When read with RDataFrame, these values will be masked

Row	R_rdf_mask_t_0	x	y	x__xVariation_0	y__xVariation_0	x__xVariation_1	y__xVariation_1
0	101b	20	-20			21	-21
1	111b	30	-30	29	-29	31	-31
2	111b	40	-40	39	-39	41	-41
3	110b	50	-50	49	-49	51	-51
4	111b	60	-60	59	-59	61	-61
5	010b	70	-70	69	-69		
6	101b	20	-20			21	-21
7	111b	30	-30	29	-29	31	-31
8	111b	40	-40	39	-39	41	-41

Missing Value API

Some datasets may have missing entries

- Spanning multiple files, snapshots with variations,
  sparse auxiliary datasets ("friends")

By default, RDataFrame stops execution if such a value is read

The missing value API (ROOT 6.34+) allows for dealing with such values

- Skip these entries using FilterAvailable
- Insert default values DefaultValueFor

x	y	aux
1	11	41
2	22	—
3	33	43
4	44	—

```
// Only histogram entries where aux is valid
auto histo_x_aux =
  df.FilterAvailable("aux")
    .Histo2D(histo2Model, "x", "aux");

// Fill in placeholders for missing values in aux
auto histo_x_defaultAux =
  df.DefaultValueFor("aux", -999)
    .Histo2D(histo2Model, "x", "aux");
```

Summary

Summary

- RDataFrame is ROOT's main entry point for analysis
- Expressing workflows as RDataFrame computations has benefits:
  - Seamlessly change input formats
  - Parallel / distributed execution with one line of code
  - Over time, improvements in RDF propagate to users, often without code changes
- Feedback on new and experimental features welcome!
  https://github.com/root-project/root

Backup

A modernized interface to ROOT files

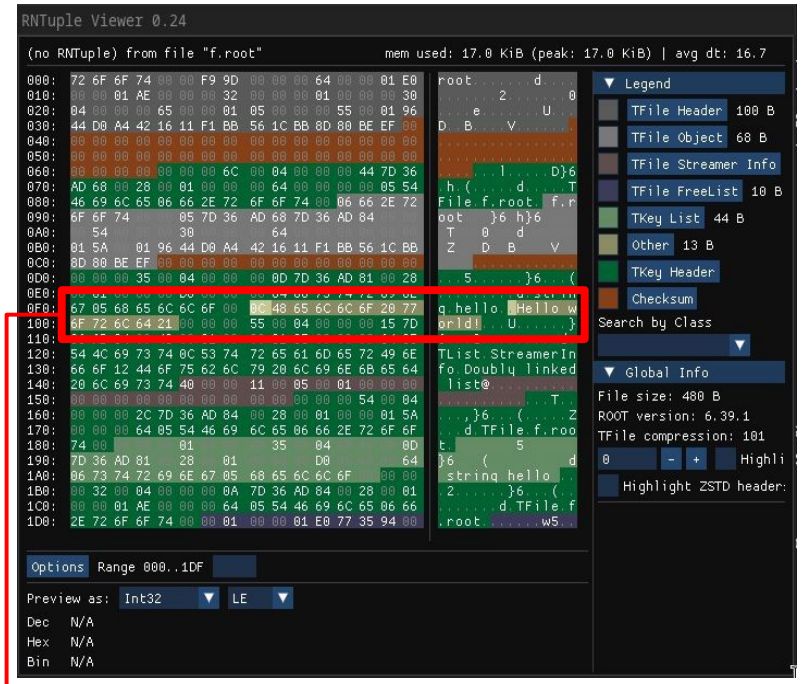
CHEP 2026
Bangkok, Thailand – 25-29 May 2026

Giacomo Parolini
ROOT Core Developers, EP-SFT, CERN

ROOT files: a refresher

- The ROOT format is one of the most widely used format to store HENP data
  - More than **2 EB** of LHC data currently stored in ROOT files
  - 10x that amount is foreseen to be gathered by HL-LHC in 2030~2040
- Binary format[1] with “filesystem-like” properties
  - Stores “objects” of arbitrary (C++) types
  - Objects are referenced by name (string)
  - Arbitrary nesting is supported through “directories”
- Support for storing custom user classes
  - Composing ROOT I/O-enabled primitives/classes
  - Custom streamers
  - Versioning, schema evolution, …

A very small ROOT file storing a single string.



`std::string hello = “Hello world!”`

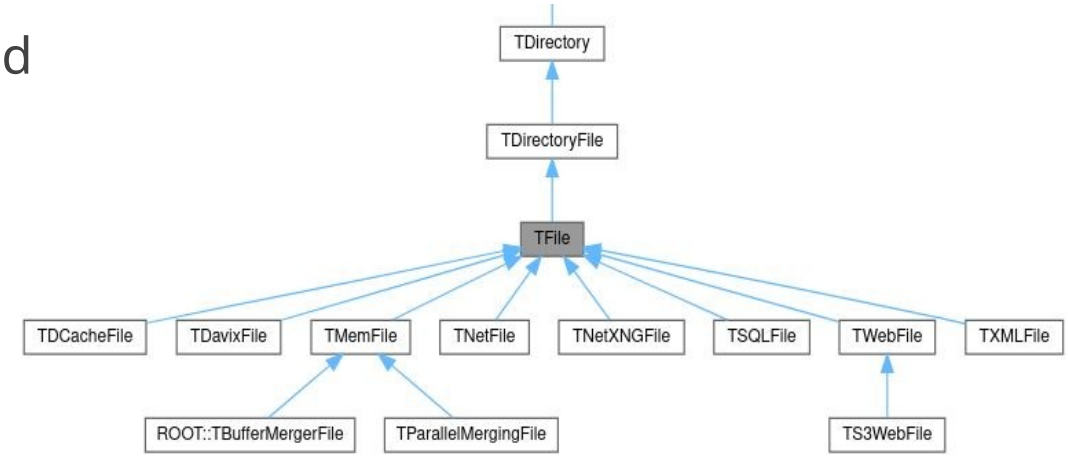
[1] non-binary formats are also supported, but niche

Table of Contents

ROOT files and TFile

- Our interface to ROOT files: **TFile**
- ...and a plethora of other related classes
- There are many operations you might want to do on a ROOT file!

➜ But this is a double-edged sword: more features ⇒ more complexity!



Can we provide a more streamlined experience for both old and new users?

Not all uses of ROOT files are the same

- Many times one “just wants to open a TFile”
  - Don’t need very sophisticated caching
  - Don’t need very optimized remote access
  - Don’t need to deal with “exotic” formats (XML, SQL, …)

- And do two things:
  - **Put** objects in the file
  - **Get** objects from the file

The entry barrier could be lower

- Mastering TFile is not trivial.
- We have a nice documentation, but:
  - Why is a TFile also a TDirectory?
  - What's a TDirectoryFile?
  - Should I use `file.WriteObject(hist)` or `hist.Write()`?
  - …

With great powers comes great subtlety

TFile has several sources of complexity/pain points:

- Interface predating smart pointers
  - object ownership not expressed by the signatures

```
TObject *TDirectoryFile::Get(const char *namecycle);
```

- Sophisticated memory management

```
// hist is now owned by the file...
TH1D *hist = file->Get<TH1D>("h");
// ...but after the following line it is owned by the caller!
hist->SetDirectory(nullptr);
```

In general we are working to minimize surprises with implicit ownership in ROOT 7

ROOT 7: Getting Ready for HL-LHC

With great powers comes great subtlety

TFile has several sources of complexity/pain points:

- Large interface, coming from decades of evolution
  - Hard to make out “basic functionality” from advanced features

Pictured: an excerpt from the TFile doc page

Introducing RFile

RFile is the new upcoming interface for ROOT files.

☞ Presented to the community at the ROOT Users Workshop 2025

Its goal:

A **minimal**, modern interface to ROOT files that is easy to use right and hard to use wrong.
An interface that guides the user, both beginner and advanced, towards success with **as little friction as possible**.

In more practical terms…

RFile should:

- Expose all necessary functionality for typical use cases (and ideally no more than that)
  - But can still handle the same scales that TFile allows for (e.g. tens of thousands of objects in a hierarchical structure)
- Prevent common mistakes
  - Presents a strongly typed interface with clear ownership
  - Doesn't do memory management "behind the user's back": every operation is explicit
  - Has consistent error handling via exceptions
- Be easily usable from Python

RFile still deals with ROOT files!

To be clear:

This is only a new **interface**. The underlying binary format remains the same!

Consequence: TFile and RFile are almost [1] fully interoperable (each can read what the other has written)

[1] exceptions (largely corner cases) will be shown in a later slide

Table of Contents

What does RFile look like?

- RFile is basically a key-value store for objects serialized in the ROOT format. It provides, at its core, these capabilities:
    1. **Create** a new RFile
    2. **Open** an existing RFile (local or remote) for reading and/or writing
    3. **Put** an object into the RFile
    4. **Get** that object back
    5. **Inspect** the content of an RFile

- And has the following properties:
    1. No implicit ownership of objects
    2. No registration to a global list of files
    3. Full backward compatibility with the current ROOT format

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

```
//// Reading
{
  unique_ptr<RFile> file = RFile::Open("file.root");
  // From top-level dir:
  unique_ptr<TH1D> hist = file->Get<TH1D>("hist");
  // From sub dir:
  unique_ptr<TH1D> subhst = file->Get<TH1D>("my/path/hist");
}
//// Writing
{
  unique_ptr<RFile> file = RFile::Recreate("file.root");
  TH1D hist;
  // To top-level dir:
  file->Put("hist", hist);
  // To sub dir:
  file->Put("my/path/hist", hist);
}
//// Query
for (RKeyInfo key : file->ListKeys("my")) {
  // ...
}
```

- RFile interactions happen through string “paths”, similar to filesystem paths
- There is no object representing a “directory”
  - No cd(), mkdir() or similar
  - No global hidden state via gDirectory
- The concept of “directory” is still used to logically group together objects
  - Utility methods to get all objects in a directory etc

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())
```

Object Ownership

- RFile can only be created via its factory methods:
  - Recreate
  - Open
  - Update
- All of these return a `std::unique_ptr<RFile>`
- `RFile::Get()` always returns a new object via a `std::unique_ptr<T>`

```
template <typename T>
std::unique_ptr<T> Get(std::string_view path) const
```

- `RFile::Put()` takes a const ref and writes the object (no ownership change)

```
template <typename T>
void Put(std::string_view path                    , const T &obj        )
```

Object and directory queries

- ListKeys allows inspecting the structure of an RFile:

```
for (auto it : file->ListKeys("myFolder/", kListObjects))
```

RKeyInfo

- GetPath() → "dir/subdir/hist"
- GetBaseName() → "hist"
- GetClassName() → "TH1D"
- GetCycle() → 3
- GetCategory() → RKeyInfo::kObject
- ...

basePath

Only list objects under this directory
(default: "" - all objects)

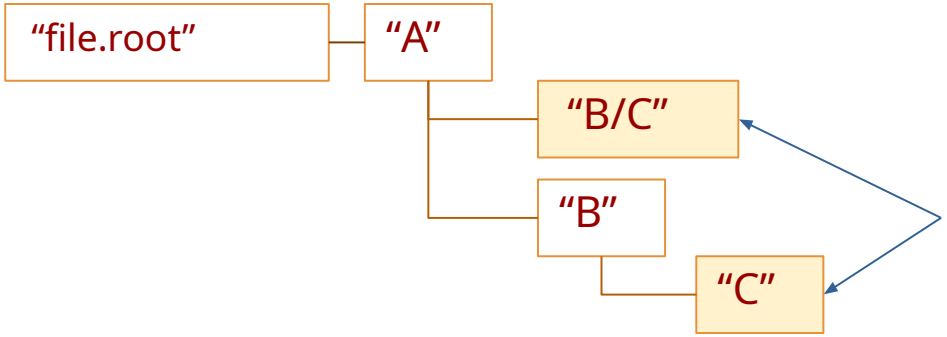
flags

kListObjects: include leaf objects' keys
kListDirs: include directories' keys
kListRecursive: recurse into subdirs

(default: kListObjects|kListRecursive)

Compatibility with TFile

- Any file written by RFile can be read by TFile
- RFile cannot read non-binary ROOT files (like XML or SQL)
- Some files written by TFile may contain objects that are inaccessible by RFile:
  - RFile can't read/write objects whose name contains '/'
  - RFile always "fully splits" the paths, meaning it will always consider "A/B/C" as "object C in directory B in directory A"



The path of both these two objects is "A/B/C": **ambiguous**!

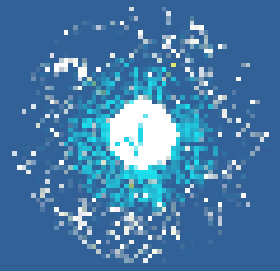
→ RFile does not allow '/' to appear in an object's basename

Conclusions

- RFile aims to be a new, alternative interface to ROOT files with a focus on usability and simplicity
  - Including from Python!
- It will **not** fully replace TFile (still useful for more advanced functionalities)
- Available **today** in the ROOT::Experimental namespace (C++ and Python)

→ out of Experimental **this November** in ROOT 6.42

We are looking for your feedback and opinions on how the API should evolve!



ROOT's New Histograms

Jonas Hahnfeld[1,2] Stephan Hageboeck[1] Jakob Blomer[1] Thorsten Kollegger[2]
jonas.hahnfeld@cern.ch

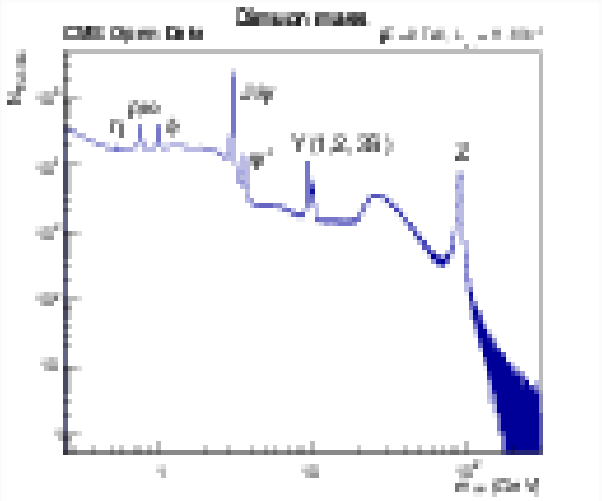
[1] CERN, Geneva, Switzerland

[2] Goethe University Frankfurt, Institute of Computer Science, Frankfurt, Germany

CHEP 2026 – May 27, 2026

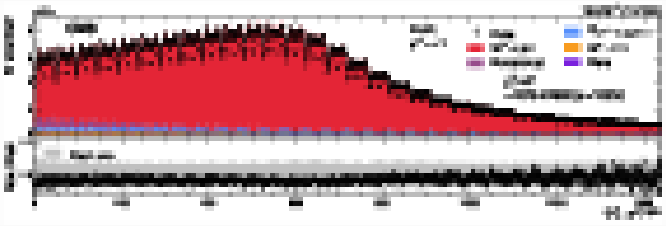
This work has been sponsored by the Wolfgang Gentner Programme of the German Federal Ministry of Education and Research (grant 13E18CHA).

Histograms in High Energy Physics



- Statistical interpretation of HEP data
  - Visualization and publication-grade plots
- Data structure for computation
  - Filling entries, optional weight parameter
  - Adding / merging, scaling
  - Projection, rebinning, slicing
  - Fitting, statistical tests: Anderson-Darling, χ^2, Kolmogorov
- This talk: histogram package for ROOT 7
  - Separating data structure from visualization and fitting / tests
  - Smooth transition via conversion functions to `TH1`, `TH2`, `TH3`, `THn`

Towards ROOT 7: Requirements for Histograms in HEP



- Multidimensional histograms
  - Example: CMS measurement of W boson mass[1]
  - Nominal histogram: 5D (see technical presentation[2])
  - Motivation for concurrent filling to reduce memory
- Global histogram statistics (optional)
  - Number of entries, sum of weights, mean and standard deviation of unbinned values
- Serialization of histograms: store and load to / from file
- Extended histogram classes: profile histograms, sparse histograms

[1] CMS-SMP-23-002

[2] https://indico.cern.ch/event/1464211/

Existing Histogram Packages (1/2)

- ROOT: `TH1`, `TH2`, `TH3`, `TProfile`, `THn`, `THnSparse`
  - Subclasses for different bin content types
  - Widely used in HEP community: analyses, DQM, ...
  - No user-defined bin content type, no concurrent filling

- GNU Scientific Library (link)
  - Only up to two dimensions, only floating-point bin content type
  - Weighted filling supported, but no error per bin

Existing Histogram Packages (2/2)

- YODA (link)
  - Developed for Monte Carlo event generator validation
  - Templated on number of dimensions and axis types
  - Provides first- and second-order statistical moments per bin

- Boost.Histogram (link)
  - Templated on axes and storage object
  - Static: axis types fully specified at compile-time
  - Usable from Python via two Scikit-HEP packages: scikit-hep/boost-histogram and scikit-hep/hist
  - Support for thread-safe storage and accumulators

Boost Histogram

Modern Histograms in ROOT 7

- Templated on bin content type, dynamic axis types at run-time
  - Reduces type explosion, helps with serialization
- Variadic user interfaces
  - Convenient user interfaces with variadic template functions
  - Also available from Python, especially for analysis workflows
- Reusable histogram components
  - Separation of classes: axis types, histogram statistics
  - `RHistEngine`: binned data without global histogram statistics
  - `RHist`: including global histogram statistics

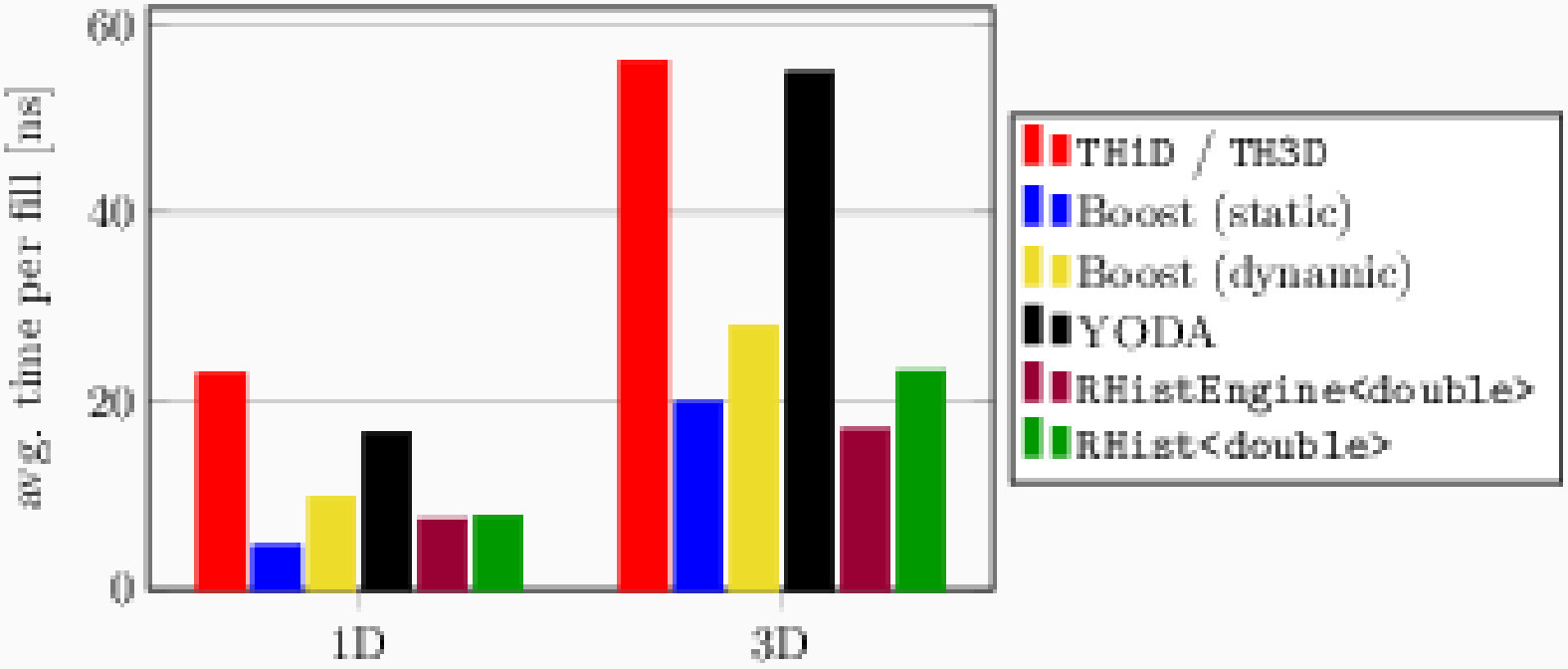
Modern Histograms in ROOT 7: Examples

```cpp
// Create a histogram with 10 normal bins in the interval [5, 15)
ROOT::Experimental::RHist<double> hist(10, {5, 15});

// ... or more explicitly by creating an axis first:
ROOT::Experimental::RRegularAxis axis(10, {5, 15});
ROOT::Experimental::RHist<double> hist(axis);

// Create a multidimensional histogram from axis objects
ROOT::Experimental::RHist<double> hist2(axis, axis);

// Convert to TH1D (returns std::unique_ptr)
auto h1 = ROOT::Experimental::Hist::ConvertToTH1D(hist);
```

Modern Histograms in ROOT 7: Examples

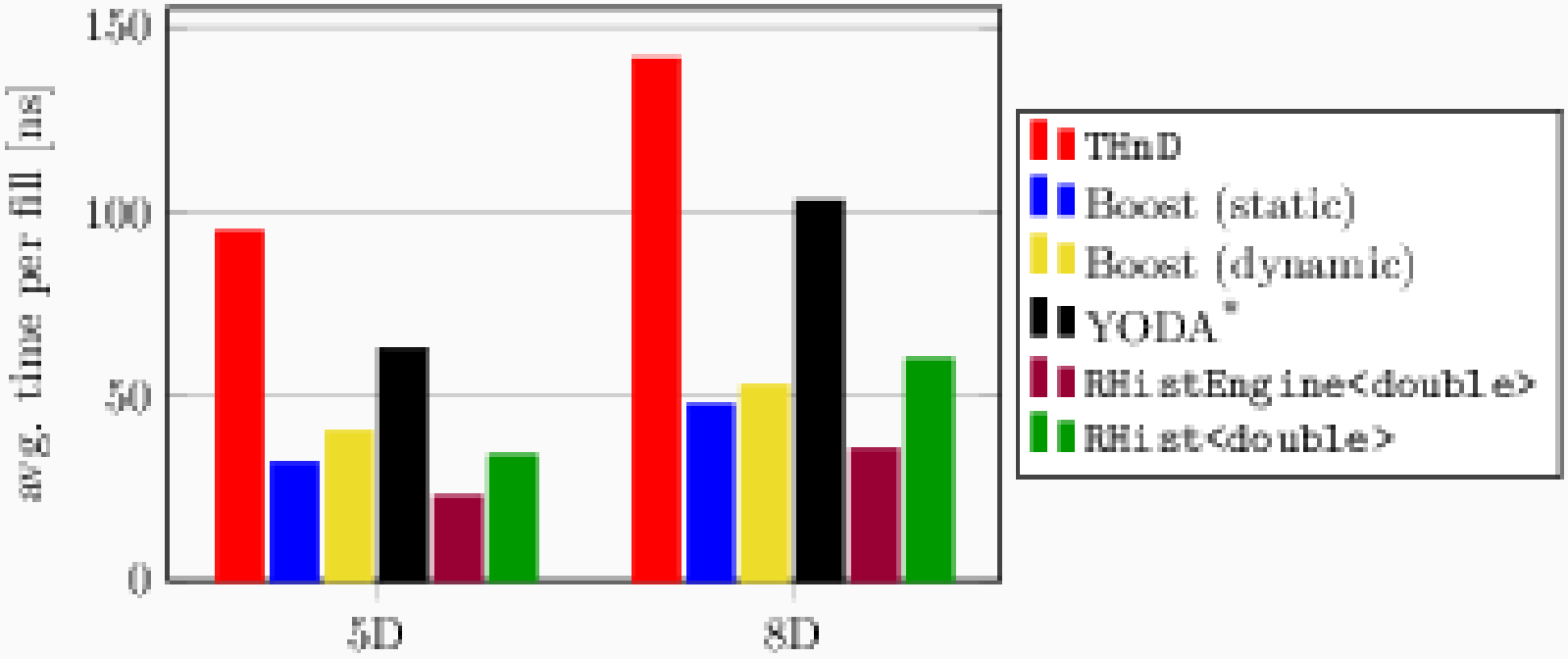
```
// Fill the one-dimensional histogram
hist.Fill(8.5);
hist.Fill(8.5, ROOT::Experimental::RWeight(0.8));

// Fill the multidimensional histogram
hist2.Fill(8.5, 9.5);
// ... or from a std::tuple:
hist2.Fill(std::make_tuple(8.5, 9.5));

// Get the bin content, returns 1.8:
/* ... */ = hist.GetBinContent(ROOT::Experimental::RBinIndex(3));
// ... or implicitly constructed also works
/* ... */ = hist.GetBinContent(3);
```

Modern Histograms in ROOT 7: Performance



Filling 4096 entries into 1D- and 3D-histogram.
Keep in mind: microbenchmarks do not reflect application performance!
Different implementations track more (YODA) or less (Boost) statistic information.

Performance of Histogram Filling



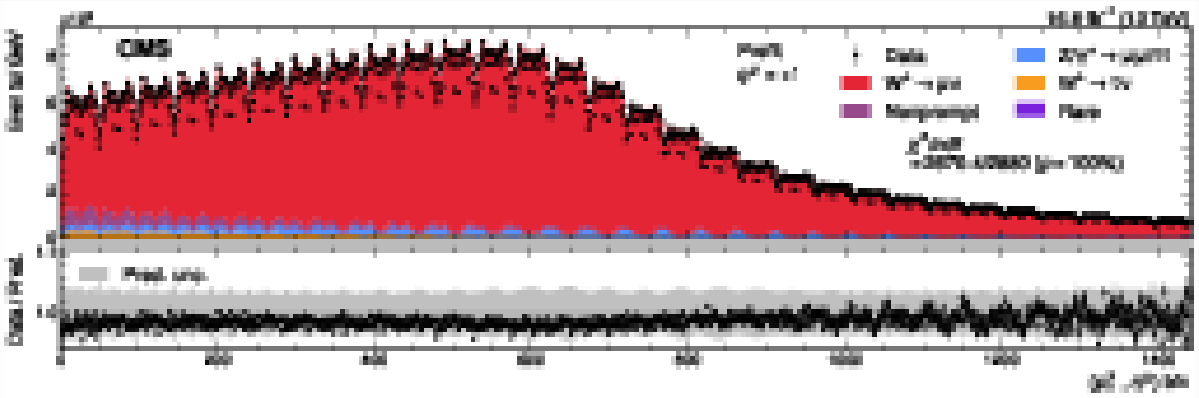
Filling 4096 entries into 5D- and 8D-histogram.

Keep in mind: microbenchmarks do not reflect application performance!

* double per bin for YODA, full statistic information did not fit into memory.

Motivation for Concurrent Filling

- Filling in multithreaded applications: one histogram per thread, merged at the end
  - Gets prohibitive for very large histograms with many threads
  - Example: CMS measurement of W mass, several GBs of histograms!



CMS-SMP-23-002

- Alternative: histogram(s) filled from multiple threads
  - Using atomic instructions, acceptable overhead with occasional contention
  - Sometimes necessary to access histograms during filling (for example DQM)
  - Conceptual work done to offer "consistent snapshots" in the future

Modern Histograms in ROOT 7: Concurrent Filling

- Concurrent filling built into the design, not an afterthought
  - Available without change of bin content type
  - Filling with atomic instructions, sequential post-processing

- `RHistEngine` has `FillAtomic` methods

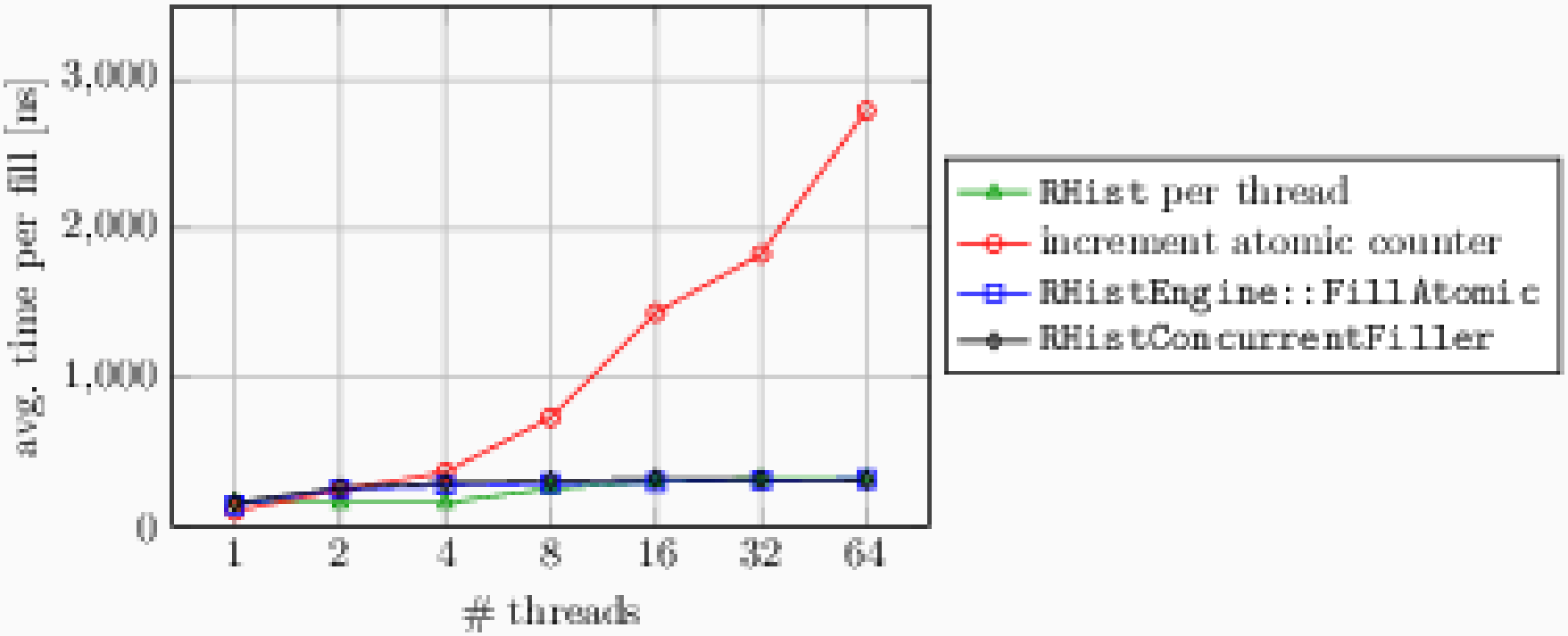
```
ROOT::Experimental::RHistEngine<double> engine(axis);
engine.FillAtomic(8.5);
engine.FillAtomic(8.5, ROOT::Experimental::RWeight(0.8));
```

Modern Histograms in ROOT 7: Concurrent Filling

- Concurrent filling built into the design, not an afterthought
  - Available without change of bin content type
  - Filling with atomic instructions, sequential post-processing
- For RHist: create RHistConcurrentFiller and RHistFillContext
  - Efficient accumulation of global histogram statistics

```
auto hist = std::make_shared<ROOT::Experimental::RHist<double>>(axis);
ROOT::Experimental::RHistConcurrentFiller filler(hist);
// per thread, reuse for many fill operations:
auto context = filler.CreateFillContext();
context->Fill(0.5);
context->Fill(0.5, ROOT::Experimental::RWeight(0.8));
```

Concurrent Filling: Performance



Microbenchmark, filling 100 million entries per thread into 3D histogram.
With perfect scalability, the time would be constant = no overhead.
Contention for single atomic counter leading to slowdown with more threads.

Integration into RDataFrame

- Concurrent filling seamlessly integrated into RDataFrame (see talk on Monday)
  - Possible to specify column types for fully compiled computation graph
  - Otherwise types inferred and code passed to JIT

```
auto hist1 = df.Hist(10, {5.0, 15.0}, "x");
auto hist2 = df.Hist({axis1, axis2}, {"x", "y"}, "w");
```

- Can fill existing histogram (via std::shared_ptr)

```
using RHistD = ROOT::Experimental::RHist<double>;
auto hist = std::make_shared<RHistD>(axis1, axis2);
auto resPtr = df.Hist(hist, {"x", "y"}, "w");
```

Histogram Operations

```cpp
// Add histogram with identical axes configuration, used for merging
hist.Add(other);
// Create a clone, independent of the original object
auto clone = hist.Clone();
// Scale all entries in the histogram
hist.Scale(0.8);

using ROOT::Experimental::RSliceSpec;
// Slice dimension to 4 normal bins
auto sliced = hist.Slice(axis.GetNormalRange(1, 5));
// Rebin axis, merging groups of two normal bins
auto rebinned = hist.Slice(RSliceSpec::ROperationRebin(2));
// Project histogram to one dimension
auto hist1 = hist2.Slice(RSliceSpec{}, RSliceSpec::ROperationSum{});
```

- **Questions**
  - *Portability*
    - ▻ Does the analysis depend on **where it runs** or **where it stores data**?
    - ▻ It should **not**
  - *Reproducibility*
    - ▻ A Student / PostDoc is leaving soon ... can someone else run the analysis?
    - ▻ Often **not** the case

- **Familiar situations**
  - "We couldn't produce updates, our local cluster is down for maintenance."
  - "We need to run things again, we forgot to change some paths in script XYZ."
  - "No updates from my side, I had to do job sitting the whole week ..."

- From personal analysis experience
  - ⅔ of time required for technicalities, ⅓ left for physics
  - → **Physics output doubled if it was the other way round?**

Usage of RDataFrame in Python

Distributed RDataFrame

- Distributed RDataFrame is now out of **Experimental** namespace
  - constructing distributed RDF became as simple as:

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

- Functionality-wise, distributed and local RDFs are now equivalent
- Several features have been ported to distributed RDataFrame:
  RDatasetSpec, C++ code inclusion and more
  👉 CHEP 2024 presentation

RDataFrame's Pythonic API So Far

- So far, one could pass a C++ expression as a string to `Define` and `Filter` operations
- Request from several communities: Can one pass Python functions?

Previously:

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

rdf2 = rdf.Define("mass", "sqrt(E*E - (px*px + py*py + pz*pz))")

rdf3 = rdf2.Filter("mass > 10")
```

RDataFrame Pythonic API Today

👉 Arbitrary Python Execution in C++

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

```python
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`

```python
@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

Новые компоненты ROOT7

- Графика — RPad / RCanvas
- Обработка данных — RDataFrame, RNTuple
- Ввод/Вывод — RFile
- Интерфейс к ML — RDataLoader