RDataFrame: status and plans

Enrico Guiraud for the ROOT team
ROOT users workshop, 9/5/2022

In this talk

- RDataFrame fundamentals
- Recently added features
- Plans for the future

RDataFrame fundamentals

HEP data processing in a nutshell

Data sources

TTree, TChain
RNTuple
Numpy Arrays
CSV
Apache Arrow
SQLite

Derived quantities

px	py	pz	eta	my_px
[1,2,3]				[2,3]
[4,5,6]				[4,6]
[7,8,9]				

Event selection

Aggregations

- histograms, profiles
- new ROOT files
- cut-flow reports
- data reductions (mean, sum,..)
- any user-defined operation

HEP data processing in a nutshell

Data sources

TTree, TChain
RNTuple
Numpy Arrays
CSV
Apache Arrow
SQLite

Derived quantities

px	py	pz	eta	my_px
[1,2,3]				[2,3]
[4,5,6]				[4,6]
[7,8,9]				

Event selection

Aggregations

- histograms, profiles
- new ROOT files
- cut-flow reports
- data reductions (mean, sum,..)
- any user-defined operation

Some aspects particular to HEP

Input datasets are much larger than memory, entries are statistically independent.

Histograms, new ROOT files as common aggregations.

Collections are ubiquitous.

HEP data processing in a nutshell

Data sources

TTree, TChain
RNTuple
Numpy Arrays
CSV
Apache Arrow
SQLite

→

Derived quantities

px	py	pz	eta	my_px
[1,2,3]				[2,3]
[4,5,6]				[4,6]
[7,8,9]				

Event selection

→

Aggregations

- histograms, profiles
- new ROOT files
- cut-flow reports
- data reductions (mean, sum,..)
- any user-defined operation

The goal

Ease of use, good performance and scaling from 1 to 1000+ cores out of the box. Extensibility.

Ergonomic support for common HEP use cases (systematics, working with collections, ...).

The HEP analysis landscape as we see it

Analysis language

⬇ ~50% □ C++
⬆ ~50% □ Python

Storage

local disk
fast-access network storage
EOS or other not-so-fast backend

Platform

laptop or PC

many-core machine

computing cluster
+ job submission

The HEP analysis landscape as we see it

Analysis language

⬇ ~50% C++
⬆ ~50% Python

Storage

local disk
fast-access network storage
EOS or other not-so-fast backend

Platform

laptop or PC | many-core machine | computing cluster + job submission

RDataFrame addresses all these use cases
with a single high-level programming model
that performs well, scales well
and enables HEP-specific ergonomics.

What RDF code looks like (C++)

```
ROOT::RDataFrame df(dataset);            // on this (ROOT, CSV, ...) dataset
auto df2 = df.Filter("x > 0")            // only accept events for which x > 0
             .Define("r2", "x*x + y*y"); // define r2 = x² + y²
auto rHist = df2.Histo1D("r2");          // plot r2 for events that pass the cut
df2.Snapshot("newtree", "out.root");     // write the skimmed data and r2 to a new ROOT file
```

What RDF code looks like (C++)

```
ROOT::RDat[aset);                                         on this (ROOT, CSV, ...) dataset
auto df2 = df.Filter("x > 0")                             [e]vents for which x > 0
             .Define("r2", "x*x + y*y");                  define r2 = x² + y²
auto rHist = df2.Histo1D("r2");                           plot r2 for events that pass the cut
df2.Snapshot("newtree", "out.root");                      write the skimmed data and r2
                                                          to a new ROOT file
```

event selection

derived quantities, object selections

data aggregations

Users can inject **arbitrary code** at all steps, which makes this relatively simple API extremely versatile.

What RDF code looks like (C++)

```
ROOT::EnableImplicitMT();
ROOT::RDataFrame df(dataset);
auto df2 = df.Filter("x > 0")
             .Define("r2", "x*x + y*y");
auto rHist = df2.Histo1D("r2");
df2.Snapshot("newtree", "out.root");
```

- `ROOT::EnableImplicitMT();` — Run a multi-thread event loop
- `ROOT::RDataFrame df(dataset);` — on this (ROOT, CSV, ...) dataset
- `auto df2 = df.Filter("x > 0")` — only accept events for which $x > 0$
- `.Define("r2", "x*x + y*y");` — define r2 = $x^2 + y^2$
- `auto rHist = df2.Histo1D("r2");` — plot r2 for events that pass the cut
- `df2.Snapshot("newtree", "out.root");` — write the skimmed data and r2 to a new ROOT file

What RDF code looks like (C++)

```
ROOT::EnableImplicitMT();
ROOT::RDataFrame df(dataset);
auto df2 = df.Filter("x > 0")
             .Define("r2", "x*x + y*y");
auto rHist = df2.Histo1D("r2");
df2.Snapshot("newtree", "out.root");
```

- `ROOT::EnableImplicitMT();` — Run a multi-thread event loop
- `ROOT::RDataFrame df(dataset);` — on this (ROOT, CSV, ...) dataset
- `auto df2 = df.Filter("x > 0")` — only accept events for which $x > 0$
- `.Define("r2", "x*x + y*y");` — define r2 = $x^2 + y^2$
- `auto rHist = df2.Histo1D("r2");` — plot r2 for events that pass the cut
- `df2.Snapshot("newtree", "out.root");` — write the skimmed data and r2 to a new ROOT file

Six lines of code! Most RDF analyses are just more of the same.
User-defined computations are naturally made modular.

What RDF code looks like (Python)

```
ROOT.EnableImplicitMT()
df = ROOT.RDataFrame(dataset)
df = df.Filter("x > 0")
       .Define("r2", "x*x + y*y")
rHist = df.Histo1D("r2")
df.Snapshot("newtree", "out.root")
```

- `ROOT.EnableImplicitMT()` ······ Run a multi-thread event loop
- `df = ROOT.RDataFrame(dataset)` ······ on this (ROOT, CSV, ...) dataset
- `df = df.Filter("x > 0")` ······ only accept events for which x > 0
- `.Define("r2", "x*x + y*y")` ······ define r2 = $x^2 + y^2$
- `rHist = df.Histo1D("r2")` ······ plot r2 for events that pass the cut
- `df.Snapshot("newtree", "out.root")` ······ write the skimmed data and r2 to a new ROOT file

PyROOT bindings make RDF easily available from Python.

Switch from TTree to RNTuple/CSV/...

```
ROOT.EnableImplicitMT()
df = MakeNTupleDataFrame(dataset)
df = df.Filter("x > 0")
        .Define("r2", "x*x + y*y")
rHist = df.Histo1D("r2")
```

- ROOT.EnableImplicitMT() .... Run a multi-thread event loop
- df = MakeNTupleDataFrame(dataset) .... on this RNTuple
- the Filter / Define / Histo1D lines .... all other code stays the same

RDataFrame enables a **seamless transition to “TTree 2.0”, RNTuple**.
The new data format is smaller and faster. Currently experimental.
See the previous talk by Jakob and Javi for more information.

Switch to distributed execution (Python)

Since v6.26 (experimental)

```
cluster = dask_jobqueue.HTCondorCluster(n_workers=64)

client = Client(cluster)

df = RDataFrame(dataset, daskclient=client)

df = df.Filter("x > 0")

        .Define("r2", "x*x + y*y")

rHist = df.Histo1D("r2")

df.Snapshot("newtree", "out.root")
```

setting up the Dask connection to HTCondor

all other code stays the same

See Vincenzo's talk later today for more on distributed RDF.

RVecs: working with collections

Select and fill: quick one-liner

```
RDataFrame("tree", "f.root").Define("pt", "muon_pt[muon_eta > 0]").Histo1D("pt").DrawClone()
```

See also the [user guide]

RVecs: working with collections

Select and fill: quick one-liner

```
RDataFrame("tree", "f.root").Define("pt", "muon_pt[muon_eta > 0]").Histo1D("pt").DrawClone()
```

Select and fill: fully compiled C++ code

```
RVecD selectPt(RVecD &pt, RVecD &eta) { return pt[eta > 0]; }

auto h = RDataFrame("tree", "f.root")
            .Define("pt", selectPt, {"muon_pt", "muon_eta"})
            .Histo1D<RVecD>("pt");
h->Draw();
```

See also the user guide

Object selection

Select some muons, plot their inv. mass

```
df.Define("m", "muon_pt > 0 && abs(muon_eta) < 2.7")
  .Define("invmass", "InvariantMass(muon_pt[m], muon_eta[m], muon_phi[m], muon_mass[m])")
  .Histo1D("invmass")
```

Sort all muon_* columns by pt

```
df.Define("sorted_idx", "Argsort(muon_pt)")
  .Redefine("muon_pt", "Take(muon_pt, sorted_idx)"
  .Redefine("muon_eta", "Take(muon_eta, sorted_idx)")
  .Redefine("muon_phi", "Take(muon_phi, sorted_idx)")
  ...
```

This does the job but it is quite verbose, so...

Object selection 2.0

Design in progress, stay tuned

Select some muons, plot their inv. mass

```
df.Define("invmass", "InvariantMass(muons[muons.pt > 0 && abs(muons.eta) < 2.7])")
  .Histo1D("invmass")
```

Sort all muons columns by pt

```
df.Redefine("muons", "SortBy(muons, muons.pt)")
```

Wide adoption from analysts

- Dark matter sensitivity study (Pani & Polesello, 2018)
- Distributed analysis with RDataFrame in TOTEM (Avati et al., 2019)
- **ATLAS**: prototype xAOD data source DOI 10.5281/zenodo.1303038
- **ALICE**: Apache Arrow support contributed by G. Eulisse
- **FCC** is developing analysis workflows based on RDF (see also the GitHub project)
- Building block in INFN analysis facility effort
- many users **“in the wild”**: 650+ threads tagged #rdataframe on the ROOT forum, about the same as #tree and #hist

RDF as a framework building block

Some examples of analysis software based on RDataFrame

- bamboo (recent talk)
- KIT's CROWN (recent talk)
- W mass analysis framework
- LoopSUSYFrame ATLAS analysis tool
- ("Latinos" CMS framework planning transition to RDF)
- narf (recent talk)
- ...

Feedback (and code) from users regularly integrated upstream (*thank you*!)

A note on performance

We care about performance. A lot.

RNTupleDS

2 K
1.50 K
1 K
500
0
ms
02/08 02/15 02/22 03/01 03/08 03/16 03/24 04/01 04/08 04/16 04/23 05/01
RNTupleDS_LHCB/25 RNTupleDS_LHCB/50 RNTupleDS_LHCB/100

Sum scalar branch

20 Mil
15 Mil
10 Mil
5 Mil
0
02/08 02/15 02/22 03/01 03/08 03/16 03/24 04/01 04/08 04/16 04/23 05/01
Just Sum After 1 Define After 5 Defines Foreach TTree TTreeReader

100 Histo Fills

20 µs
15 µs
10 µs
5 µs
0 µs
02/08 02/15 02/22 03/01 03/08 03/16 03/24 04/01 04/08 04/16 04/23 05/01
book only Avg: 5.33 µs no RDF Avg: 2.67 µs no IMT Avg: 12.6 µs range deduction, no RDF Avg: 2.64 µs range deduction, no IMT Avg: 16.2 µs

Sum vector branch

50 Mil
40 Mil
30 Mil
20 Mil
10 Mil
02/08 02/15 02/22 03/01 03/08 03/16 03/24 04/01 04/08 04/16 04/23 05/01
Just Sum After 1 Define After 5 Defines TTreeReader TBranch

100k Histo Fills

80 ms
60 ms
40 ms
20 ms
0 µs
02/08 02/15 02/22 03/01 03/08 03/16 03/24 04/01 04/08 04/16 04/23 05/01
no RDF Avg: 1.10 ms no IMT Avg: 2.92 ms range deduction, no RDF Avg: 1.06 ms range deduction, no IMT Avg: 3.18 ms IMT Avg: 63.2 ms
range deduction, IMT Avg: 4.21 ms

Jit large graphs

35 s
30 s
25 s
20 s
15 s
10 s
5 s
0 µs
02/08 02/15 02/22 03/01 03/08 03/16 03/24 04/01 04/08 04/16 04/23 05/01
100 nodes 1000 nodes 10000 nodes

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

We care about performance. A lot.

From [Avati et al., 2019]

[github.com/root-project/rootbench]

We care about performance. A lot.

Distributed RDF benchmark (dimuon, 4000x data)

We care about performance. A lot.

Fully compiled C++ RDataFrame (ROOT@db6a9d62f)

query,	1x data (s),	10x data (s)
Q1	0.37	1.50
Q2	0.46	3.70
Q3	0.73	6.23
Q4	0.65	5.92
Q5	0.84	7.45
Q6	3.08	27.99
Q7	2.56	22.27
Q8	1.17	10.22

Coffea 0.7.12 (using chunksize=2**19)

query,	1x data (s),	10x data (s)
Q1	1.40	4.24
Q2	1.51	5.76
Q3	1.81	7.96
Q4	1.65	6.58
Q5	2.41	12.43
Q6	13.89	124.59
Q7	4.19	29.12
Q8	3.27	17.70

- note that these benchmarks are not representative of large analysis workloads
- see also this ACAT talk by Nick Smith

Benchmark from github.com/nsmith-/coffea-benchmarks
Setup: AMD EPYC 7702P, using 48 physical cores, data read from filesystem cache

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

We care about performance. A lot.

NanoAOD events processed at 400 kHz when producing ~6k histograms.

zlib-compressed data read from local SSD
128 threads on 2x AMD EPYC 7742
~CMS Wmass analysis framework

“turnaround of a few hours for O(100) plots (thousands of histograms) of the CMS Run2 data on a batch system”
~bamboo

Hist Type	Hist Config	Evt. Loop	Total	CPUEff	RSS
ROOT THnD	10 x 103 x 5D	59m39s	74m05s	0.74	400GB
ROOT THnD	10 x 6D back	7m54s	25m09s	0.27	405GB
ROOT THnD	10 x 6D front	13m52s	30m27s	0.42	406GB
Boost (“sta”)	10 x 6D back	7m07s	7m17s	0.90	9GB
Boost (“sta”)	10 x 6D front	3m22	3m33s	0.86	9GB
Boost (“sta”)	10 x (5D + 1-tensor)	1m54s	2m04s	0.81	9GB
Boost (“sta”)	1 x (5D + 2-tensor)	1m32s	1m42s	0.77	9GB

From this talk by Josh Bendavid

Processing lz4-compressed ROOT data at 2 GB/sec

32 threads running on AMD Ryzen
Reading from a local NVME SSD disk
~CMS momentum correction

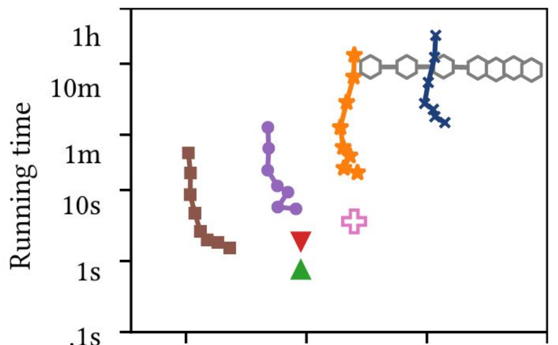
We care about performance. A lot.

Fu

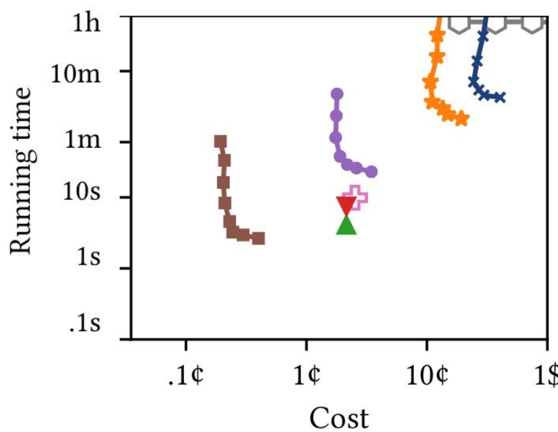
Nanc
when

zlib-c
128 t
~CMS

Hist Type
ROOT THnD
ROOT THnD
ROOT THnD
Boost ("sta")
Boost ("sta")
Boost ("sta")
Boost ("sta")



(d) (Q4)



- AsterixDB
- Athena
- BigQuery
- BigQuery (pre-loaded)
- Postgres
- Presto
- RDataFrames
- RumbleDB

"...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

(100)
of the
em"

Ryzen
SSD disk
on

Performance: the bottom line

- Given users' feedback and our own benchmarks,
  RDataFrame's C++ core enables fast turnaround for complex analyses usecases
- RDataFrame scales well to many cores, many nodes, many histograms
- Performance is always ongoing work: we are constantly looking for feedback/usecases

Recently added features

On-the-fly systematic variations

In ROOT 6.26 (experimental)

Python

```
nominal_hx =
  df.Vary("pt", "RVecD{pt*0.9, pt*1.1}",  ["down", "up"])
    .Filter("pt > k")
    .Define("x", someFunc, ["pt"])
    .Histo1D("x")

hx = ROOT.RDF.VariationsFor(nominal_hx)
hx["nominal"].Draw()
hx["pt:down"].Draw("SAME")
```

Varied columns can be used in Defines, Filters, as histogram value/weights and anything else.

Variations automatically propagate to selections, derived quantities and results.

Multi-thread and **distributed** execution **just works**.

On-the-fly systematic variations

In ROOT 6.26 (experimental)

Python

attach an up/down variation to “pt”

```python
nominal_hx =
   df.Vary("pt", "RVecD{pt*0.9, pt*1.1}",  ["down", "up"])
     .Filter("pt > k")
     .Define("x", someFunc, ["pt"])
     .Histo1D("x")

hx = ROOT.RDF.VariationsFor(nominal_hx)
hx["nominal"].Draw()
hx["pt:down"].Draw("SAME")
```

Varied columns can be used in Defines, Filters, as histogram value/weights and anything else.
Variations automatically propagate to selections, derived quantities and results.
Multi-thread and **distributed** execution **just works**.

On-the-fly systematic variations

In ROOT 6.26 (experimental)

Python

attach an up/down variation to “pt”

proceed as usual, as if working with nominal values only

```python
nominal_hx =
   df.Vary("pt", "RVecD{pt*0.9, pt*1.1}",  ["down", "up"])
     .Filter("pt > k")
     .Define("x", someFunc, ["pt"])
     .Histo1D("x")

hx = ROOT.RDF.VariationsFor(nominal_hx)
hx["nominal"].Draw()
hx["pt:down"].Draw("SAME")
```

Varied columns can be used in Defines, Filters, as histogram value/weights and anything else.
Variations automatically propagate to selections, derived quantities and results.
Multi-thread and **distributed** execution **just works**.

On-the-fly systematic variations

In ROOT 6.26 (experimental)

Python

attach an up/down variation to “pt”

```
nominal_hx =
   df.Vary("pt", "RVecD{pt*0.9, pt*1.1}",  ["down", "up"])
     .Filter("pt > k")
     .Define("x", someFunc, ["pt"])
     .Histo1D("x")

hx = ROOT.RDF.VariationsFor(nominal_hx)
hx["nominal"].Draw()
hx["pt:down"].Draw("SAME")
```

proceed as usual, as if working with nominal values only

obtain all variations

Varied columns can be used in Defines, Filters, as histogram value/weights and anything else.
Variations automatically propagate to selections, derived quantities and results.
Multi-thread and **distributed** execution **just works**.

On-the-fly systematic variations

In ROOT 6.26 (experimental)

Python

attach an up/down variation to “pt”

proceed as usual, as if working with nominal values only

```python
nominal_hx = 
   df.Vary("pt", "RVecD{pt*0.9, pt*1.1}", ["down", "up"])
     .Filter("pt > k")
     .Define("x", someFunc, ["pt"])
     .Histo1D("x")

hx = ROOT.RDF.VariationsFor(nominal_hx)
hx["nominal"].Draw()
hx["pt:down"].Draw("SAME")
```

N.B. in 6.26 the spelling will be ROOT.RDF.**Experimental**.VariationsFor

obtain all variations

Varied columns can be used in Defines, Filters, as histogram value/weights and anything else.
Variations automatically propagate to selections, derived quantities and results.
Multi-thread and **distributed** execution **just works**.

Jet energy and MET corrections

In ROOT 6.26 (experimental)

C++

```
df.Define("jetCorrection", "1.")
  .Define("METCorrection", "1.")
  .Vary({"jetCorrection", "METCorrection"},
        getJetAndMETCorrections, inputCols, {"down", "up"}, "jetAndMET")
 .Redefine("jet_E", "jet_E*jetCorrection")
 .Redefine("MET", "MET*METCorrection")
```

Calls for some syntactic sugar or a helper function, thinking in progress.

Python functions in RDF with Numba

v6.24

Python

```
@ROOT.Numba.Declare(["RVecD","RVecD"], "RVecD")
def good_pts(pts, etas): # pts and etas are NumPy arrays
   return pts[etas > 0]

df.Define("pt", "Numba::good_pts(muon_pt, muon_eta)").Histo1D("pt").DrawClone()
```

Python functions in RDF with Numba

v6.24

Python

```
@ROOT.Numba.Declare(["RVecD","RVecD"], "RVecD")
def good_pts(pts, etas): # pts and etas are NumPy arrays
    return pts[etas > 0]

df.Define("pt", "Numba::good_pts(muon_pt, muon_eta)").Histo1D("pt").DrawClone()
```

Python

```
# the code above will soon just be:
df.Define("pt", lambda muon_pt, muon_eta: muon_pt[muon_eta > 0])
```

Definition of per-sample values

v6.26

C++

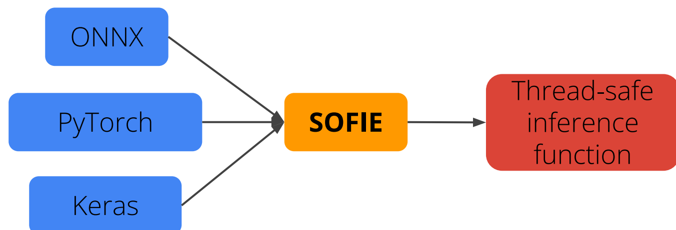
```
df.DefinePerSample("weight",
      [](unsigned slot, const RDF::RSampleInfo &s) {
            return s.Contains("MC") ? 0.5 : 1.; })
   .Histo1D("value","weight");
```

DefinePerSample evaluates a quantity that depends on the sample being processed.
Useful e.g. to define different event weights for MC and data.

See the docs.

ML inference in RDF with SOFIE

v6.26 (experimental)




```cpp
#include "Higgs_trained_model.hxx"
#include "TMVA/SOFIEHelpers.hxx"

SofieFunctor<TMVA_SOFIE_model::Session> functor(nThreads);
auto h = df.DefineSlot("y", functor, colNames).Histo1D("y");
h->Draw();
```

- Generated code only depends on BLAS/Eigen
- Combines with RDF's multi-threading
- Users can easily add custom operators

Also see [this talk by Lorenzo on Wednesday](#).

RDF ⇔ NumPy arrays

v6.18

- **TTree to NumPy** arrays via RDataFrame

```
cols = df.Filter("x > 10").AsNumpy(["x", "y"])
```

- **NumPy arrays to RDataFrame**

```
data = {"x": np.array(...), "y": np.array(...), ...}
df = ROOT.RDF.MakeNumpyDataFrame(data)
```

Work in progress: RDF ⇔ Awkward arrays, see this talk by Ianna on Tuesday.

Concluding remarks

Coming soon

- Performance improvements (e.g. bulk processing, ROOT PoW 2022)
- Collection aggregations (muon_{pt,eta,phi} → muons) being discussed
- Simpler Pythonic interfaces (less C++ strings in Python code), PoW 2022
- Allow default values for missing branches, PoW 2022, GitHub issue
- Debug symbols in jitted code (better error messages), PoW 2022, GitHub PR
- Dataset specification with user-defined sample labels
- ...and more, see our GitHub issue tracker

Designed for you, with you

RDataFrame is a battle-tested, fast, versatile interface for modern HEP analysis.

RDataFrame (and ROOT) keeps **evolving**, in **cooperation with the community**.

Your feedback is **critical to prioritize correctly**!

The docs

[RDF user guide]

[RDF tutorials]

EOF

...but wait, there is more!

Plotting a dimuon mass

Python

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

# For simplicity, select only events with exactly two muons and require opposite charge
df = df.Filter("nMuon == 2")\
       .Filter("Muon_charge[0] != Muon_charge[1]")

df = df.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.Histo1D(("dimuon_hist", "Dimuon mass;m_{#mu#mu} (GeV);N_{Events}", 30000, 0.25, 300),
               "Dimuon_mass")
```

See full tutorial

Plotting a dimuon mass

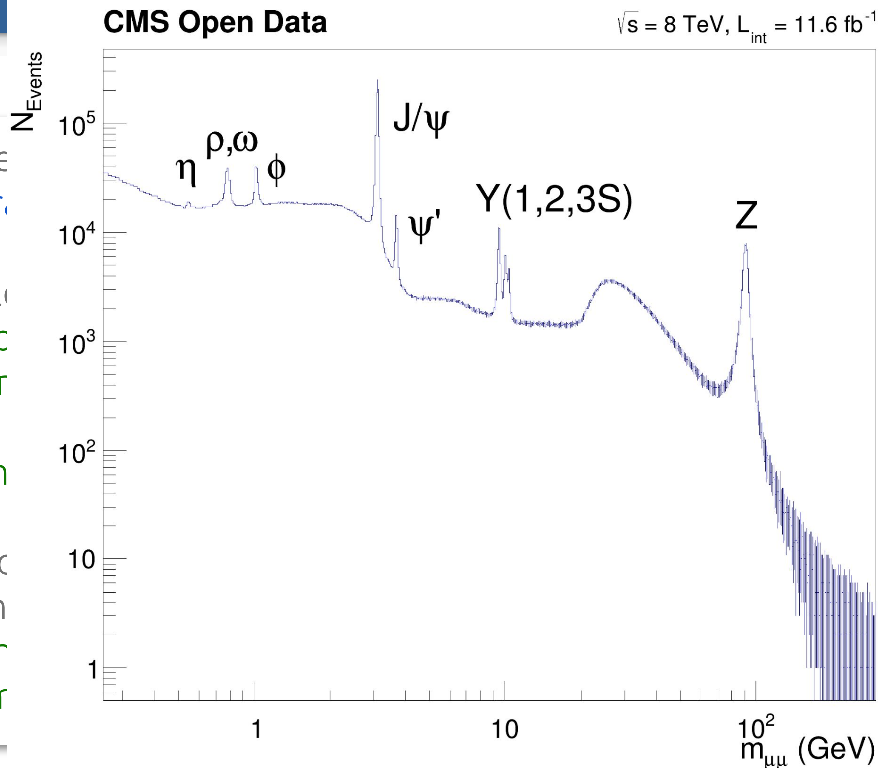
Python

```
# Create dataframe
df = ROOT.RDataFr                                    l/Run2012BC_DoubleMuParked_Muons.root")

# For simplicity, sel                                te charge
df = df.Filter("nMuc
        .Filter("Muor

df = df.Define("Dim                                  i, Muon_mass)")

# Make histogram c
# Note how we can
h = df.Histo1D(("dir                                 ', 30000, 0.25, 300),
           "Dim
```



See full tutorial

Inspecting (remote) data

```
df = ROOT.RDataFrame("Events", "root://eospublic.cern.ch//eos/opendata/cms/derived-data/AOD2NanoAODOutreachTool/Run2012BC_DoubleMuParked_Muons.root")
df.Filter("nMuon == 2").Display("Muon_.*").Print()
```

Row	Muon_charge	Muon_eta	Muon_mass	Muon_phi	Muon_pt
0	-1	1.06683f	0.105658f	-0.0342727f	10.7637f
	-1	-0.563787f	0.105658f	2.54262f	15.7365f
1	1	-0.427780f	0.105658f	-0.274792f	10.5385f
	-1	0.349225f	0.105658f	2.53978f	16.3271f
6	-1	-0.532089f	0.105658f	-0.0717980f	57.6067f
	1	-1.00417f	0.105658f	3.08952f	53.0451f
7	1	-0.771659f	0.105658f	-2.24527f	11.3197f
	-1	-0.700997f	0.105658f	-2.18096f	23.9064f
8	-1	0.441807f	0.105658f	0.677852f	10.1936f
	1	0.702117f	0.105658f	-2.03440f	14.2041f

See docs

TTree friends, TTree "joins"

```
TTree mainTree = ...;
TTree auxTree = ...;

auxTree.BuildIndex("Run", "Event");
mainTree.AddFriend(&auxTree);

auto df = ROOT::RDataFrame(mainTree);
```

RDataFrame will detect the input trees' friends, TEntryLists, *indexed* friends (simple joins) and make their columns available.

We are working on a simpler API to specify input datasets.

Creating RooFit datasets with RDF

```
RooRealVar x("x", "x", -5., 5.);
RooRealVar y("y", "y", -50., 50.);
auto myDataSet = df.Book<double, double>(
    RooDataSetHelper{"dataset",                 // Name
                     "Title of dataset",        // Title
                     RooArgSet(x, y) },         // Variables to create in dataset
                     {"x", "y"}                 // Column names from RDataFrame
);
```

See the docs

Cutflow reports with RDF

```
df.Filter("x > 0", "xcut").Filter("y < 2", "ycut");
df.Report().Print();
```

```
// output
xcut  : pass=25 all=50 -- eff=50.00 % cumulative eff=50.00 %
ycut  : pass=23 all=25 -- eff=92.00 % cumulative eff=46.00 %
```

Report provides statistics for all filters *with a name.*

Stats can be printed or inspected programmatically.

Modularity comes in two flavors

Factoring out RDF operations

```
def apply_cuts(df):
  df = df.Filter(...).Filter(...)
  return df

df = apply_cuts(df)
```

Factoring out user-defined logic

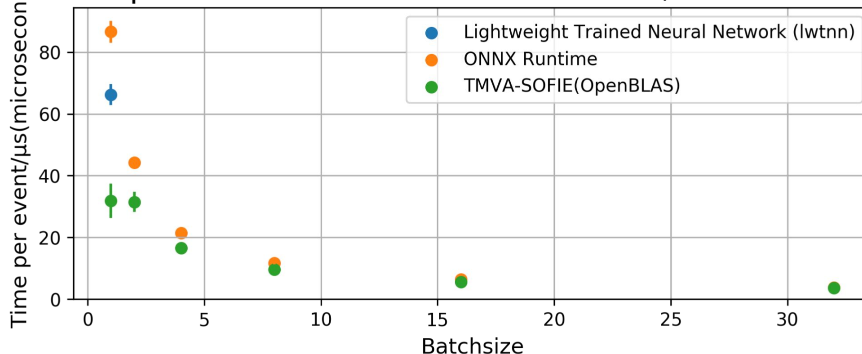
```
ROOT.gInterpreter.Declare("""
RVecD  EvalX(RVecD& x, RVecD& y) { … }
""")

df = df.Define("x", ROOT.EvalX, input_columns)
```

Combined, these patterns naturally isolate complexity,
keep analysis code clean, make components reusable.
Using Python for steering and C++ for a fast inner loop.

TMVA SOFIE: performance benchmarks

Deep neural network



Convolutional neural network

Model	SOFIE/ms	ONNXRuntime/ms
1xConv, Batch=1	**0.05**	0.08
14xConv, Batch=1	126	**100**
14xConv, Batch=32	50	**49**
Resnet18	44	**34**

Varying multiple columns together

```
C++
df.Vary({"pt", "eta"},
      "RVec<RVecF>{{pt*0.9, pt*1.1}, {eta*0.9, eta*1.1}}",
      /*variationTags=*/{"down", "up"},
      /*variationName=*/"ptAndEta")
```

- will produce 3 “universes”: nominal, ptAndEta:down, ptAndEta:up
- "pt" and "eta" will vary in lockstep rather than one at a time
- looking into simplifying `RVec<RVecF>`

Varying multiple columns together

the expression returns an array of values *per column*

```
C++
df.Vary({"pt", "eta"},
        "RVec<RVecF>{{pt*0.9, pt*1.1}, {eta*0.9, eta*1.1}}",
        /*variationTags=*/{"down", "up"},
        /*variationName=*/"ptAndEta")
```

- will produce 3 “universes”: nominal, ptAndEta:down, ptAndEta:up
- "pt" and "eta" will vary in lockstep rather than one at a time
- looking into simplifying `RVec<RVecF>`

Multiple variations

C++

```
auto df = _df.Vary("pt",
                   "RVecD{pt*0.9, pt*1.1}",
                   {"down", "up"})
             .Vary("eta",
                   [](float eta) { return RVecF{eta*0.9, eta*1.1}; },
                   {"eta"},
                   /*nVariations=*/2);

auto nom_h = df.Histo2D("pt", "eta");
auto all_h = ROOT::RDF::VariationsFor(nom_h);
```

Variations are applied one at a time:
the code above creates “universes” nominal, pt:down, pt:up, eta:0, eta:1.

Vary expressions use any columns

C++

```
df.Vary("pt",
        [](float ptdown, float ptup) { return RVecF{ptdown, ptup}; },
        {"frienddown.pt", "friendup.pt"},
        /*variationTags=*/{"down", "up"});
```

Here we evaluate the varied values of “pt” from columns “frienddown.pt” and “friendup.pt”.

Similarly we could evaluate variations for histogram weights
as an **arbitrary function of any other column values** or other objects.

RNTuple: improving on TTree

Modern TTree successor in terms of on-disk format and low-level software API.

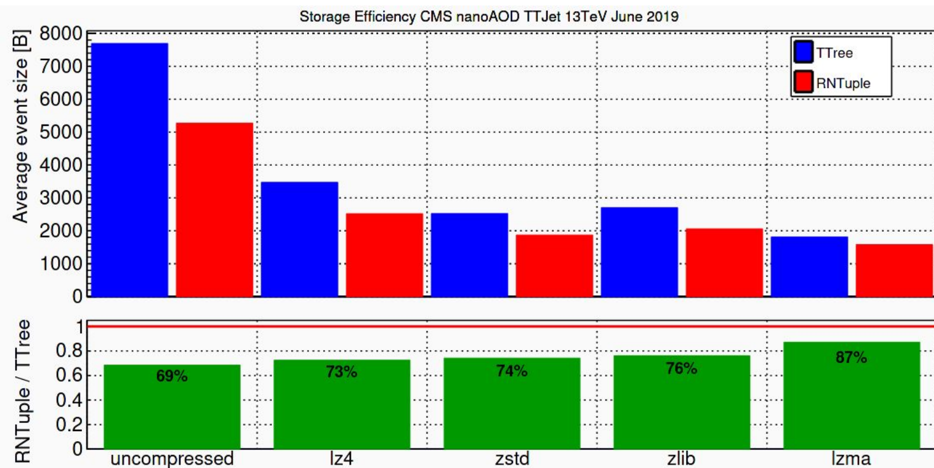
Why a redesign?

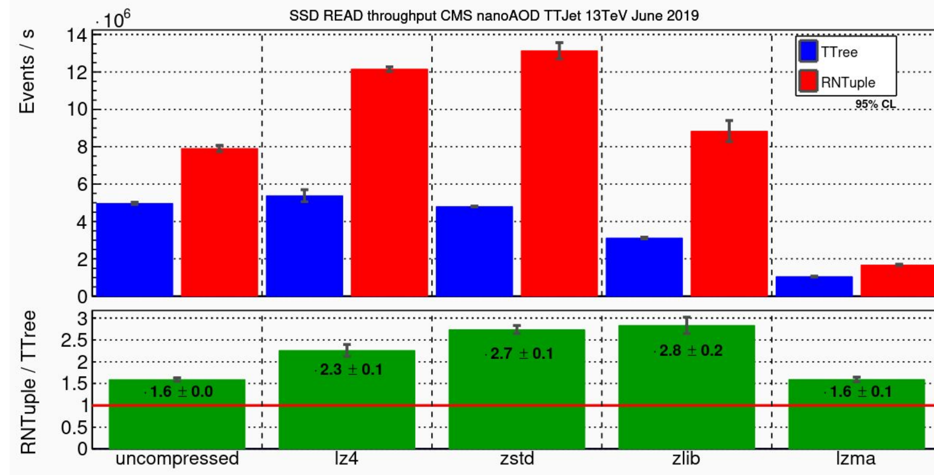
- less disk and CPU usage for same data content
- lossy compression, accelerated data-specific/-optimized algorithms
- native support for object stores (targeting HPC)
- systematic use of exceptions to prevent silent I/O errors

Seamless transition for users thanks to RDataFrame.

We see it as a Run 4 technology, in the experimental state for Run 3.

RNTuple: smaller, faster than TTree





Goal

10 GB/s throughput from SSD/fast network to histogram on a single machine.

[Blomer et al., "Evolution of the ROOT Tree I/O", CHEP 2019]

RNTuple: the fastest HEP I/O

J. Lopez Gomez, RNTuple: Performance, Updates and Outlook for 2022

Transparent optimizations

Open data benchmarks, queries 1 and 7 (data read from warm cache, 64 physical cores)

Continuous work on performance: faster analysis with every ROOT release, transparently.
Special thanks to J. Bendavid for his contributions in this domain.

DistRDF benchmark setup: cluster

Accessed HPC cluster at CERN

- Dask: launch jobs using `SSHCluster`
- Spark: reserve nodes and launch **standalone** cluster
- **Scheduler** runs in a **cluster node**, not on client
- Test from 128 cores to 2048 cores (32 cores per node)

DistRDF benchmark setup: dataset

- Dimuon analysis
- 4000x original dataset
  - **8800 GB**
  - ZLIB compressed
- Data is **node-local** during analysis
- **100%** read and processed in the analysis
- Time to plot at 2048 cores: ~100s