Simpler, faster analysis with modern ROOT

Enrico Guiraud, I. Kabadzhov , A. Naumann, V. Padulano, Pawan J., E. Tejedor
for the ROOT team
ICHEP 2022, 8/7/2022

The HEP analysis landscape as we see it

not covered here, see RooFit talk!

Analysis life cycle

skimming, ntuple production	quick exploration, first implementation	systematics, scale out	statistical analysis

Platforms

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

Analysis languages

- ~50% C++
- ~50% Python

Storage

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

A swiss-army knife for data analysis

Analysis life cycle

skimming, ntuple production — quick exploration, first implementation — systematics, scale out

Platforms

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

Analysis languages

⬇ ~50% C++
⬆ ~50% Python

Storage

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

ROOT.RDataFrame is a modern analysis interface that addresses all these use cases with **one high-level programming** model that performs well, scales well and enables **HEP-specific ergonomics**, in C++ and Python.

What RDF code looks like (Python)

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

- `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 = x² + y²
- `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

What RDF code looks like (Python)

df = ROOT.R ... ataset)	on this (ROOT, CSV, ...) dataset
df = df.Filter("x > 0")	ept events for which x > 0
.Define("r2", "x*x + y*y")	define r2 = x² + y²
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

event selection

derived quantities, object selections

data aggregations

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

Switch on multi-threading (Python)

`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

Switch to distributed execution (Python)

Since v6.26 (experimental)

```
cluster = dask_jobqueue.HTCondorCluster(n_workers=64)
df = RDataFrame(dataset, daskclient=Client(cluster))
df = df.Filter("x > 0")
       .Define("r2", "x*x + y*y")
rHist = df.Histo1D("r2")
df.Snapshot("newtree", "out.root")
```

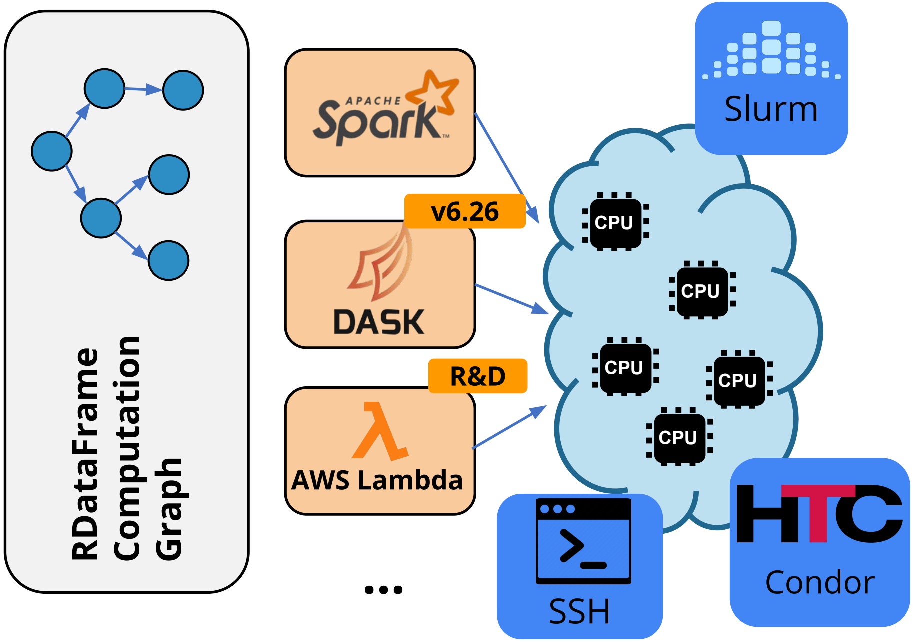
connect to HTCondor via Dask

other code stays the same

Also see [this tutorial](), [the docs](), the [recent ATTF talk]()

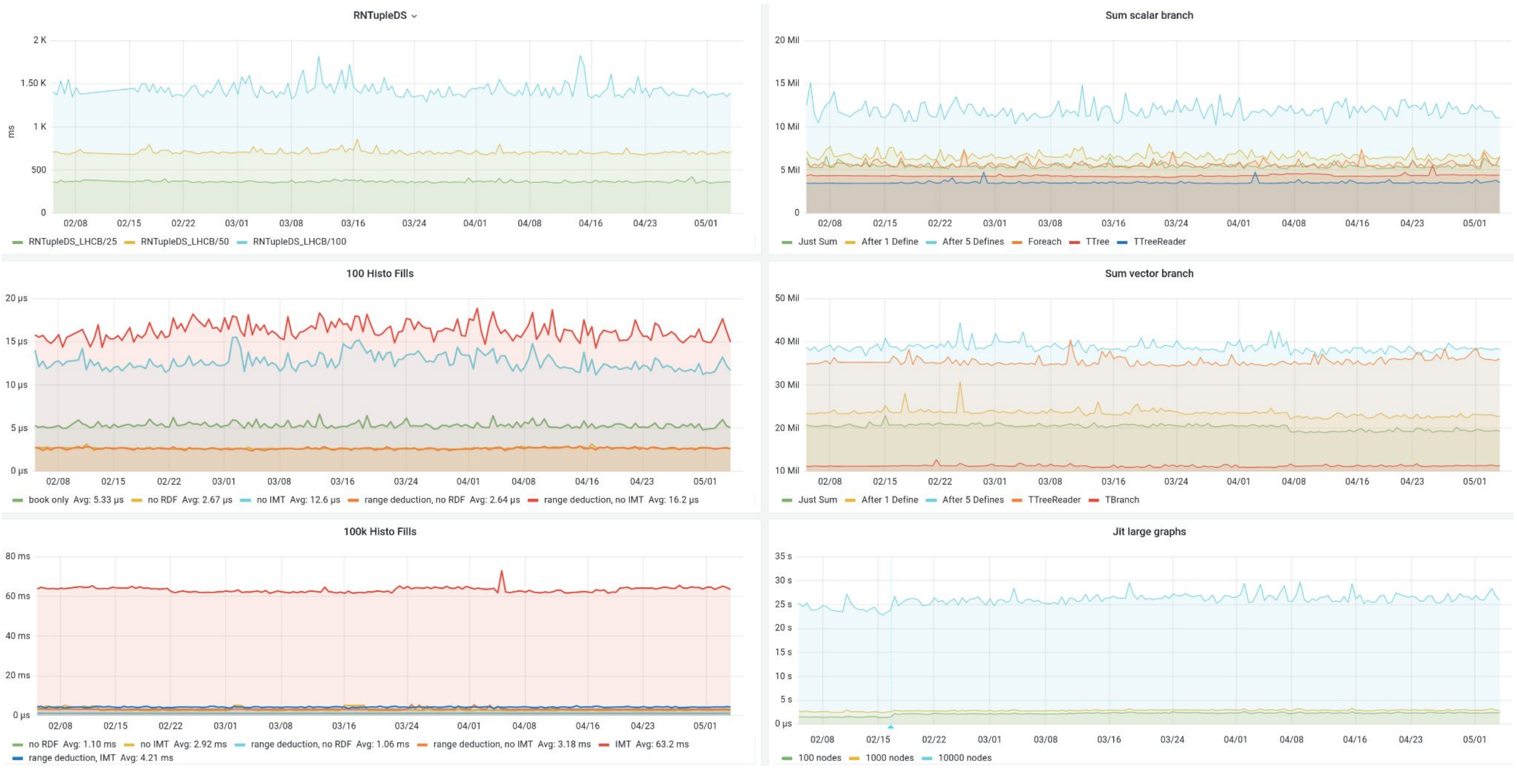
Distributed execution with RDataFrame

- Enables **interactive large-scale distributed** data analysis
- Python RDF API, C++ event loop
- Full access to ROOT I/O
- Let Spark/Dask/HTCondor/Slurm/SSH…. take care of scheduling and resource management
- Transparently merges results coming from different computing nodes



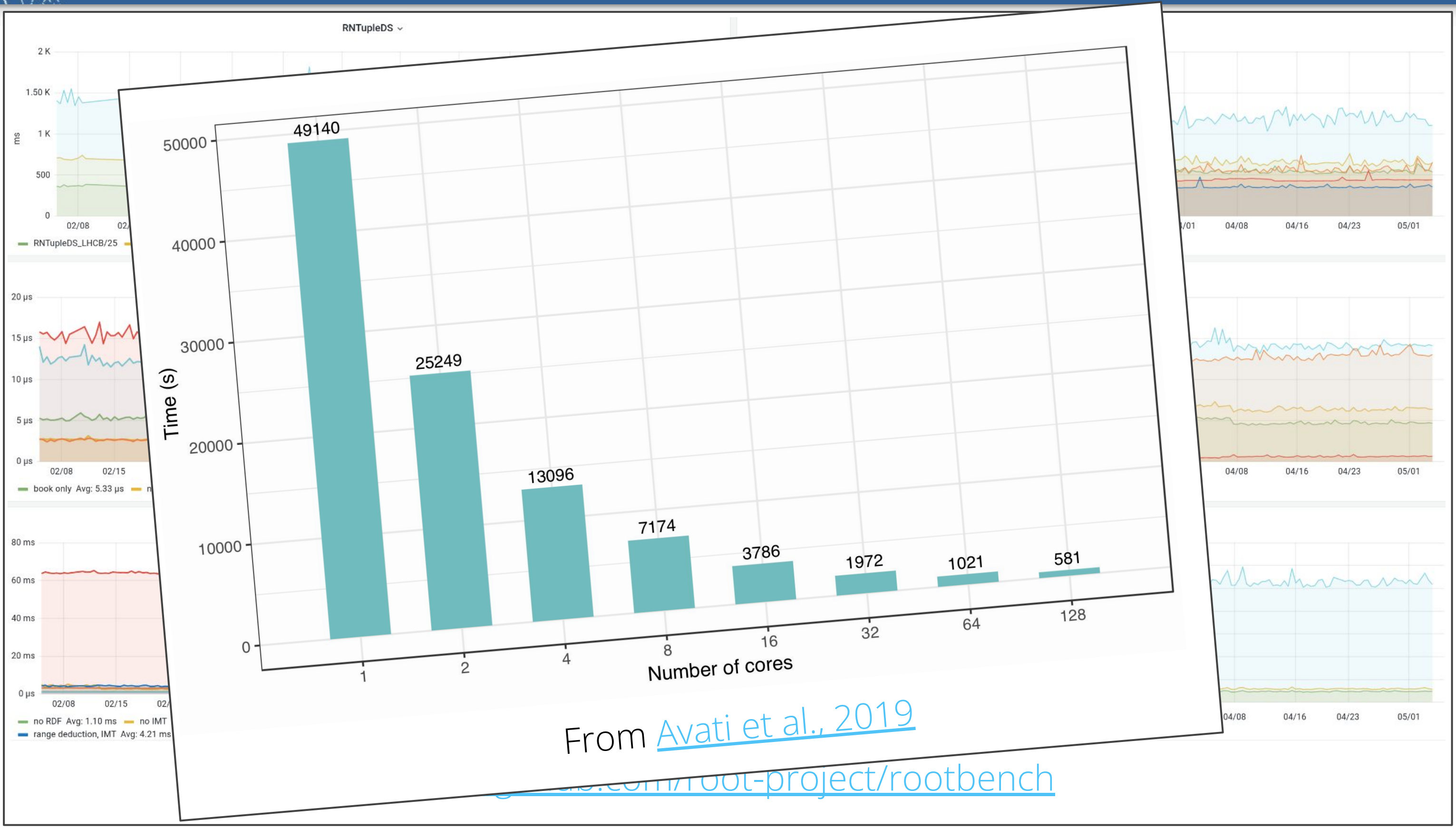
A note on performance

We care about performance. A lot.



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

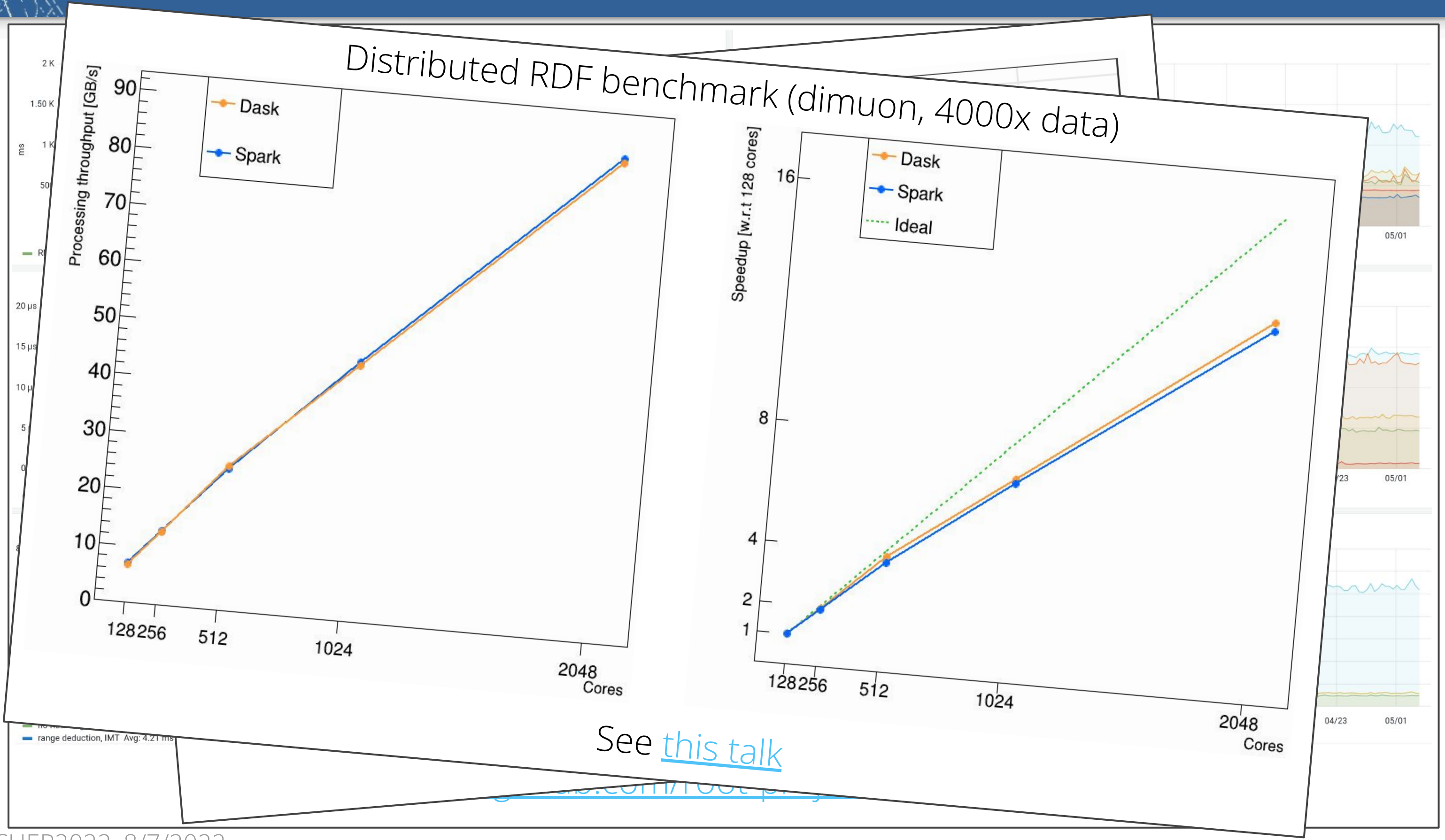
We care about performance. A lot.



From Avati et al., 2019

ub.com/root-project/rootbench

We care about performance. A lot.



See [this talk]()

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

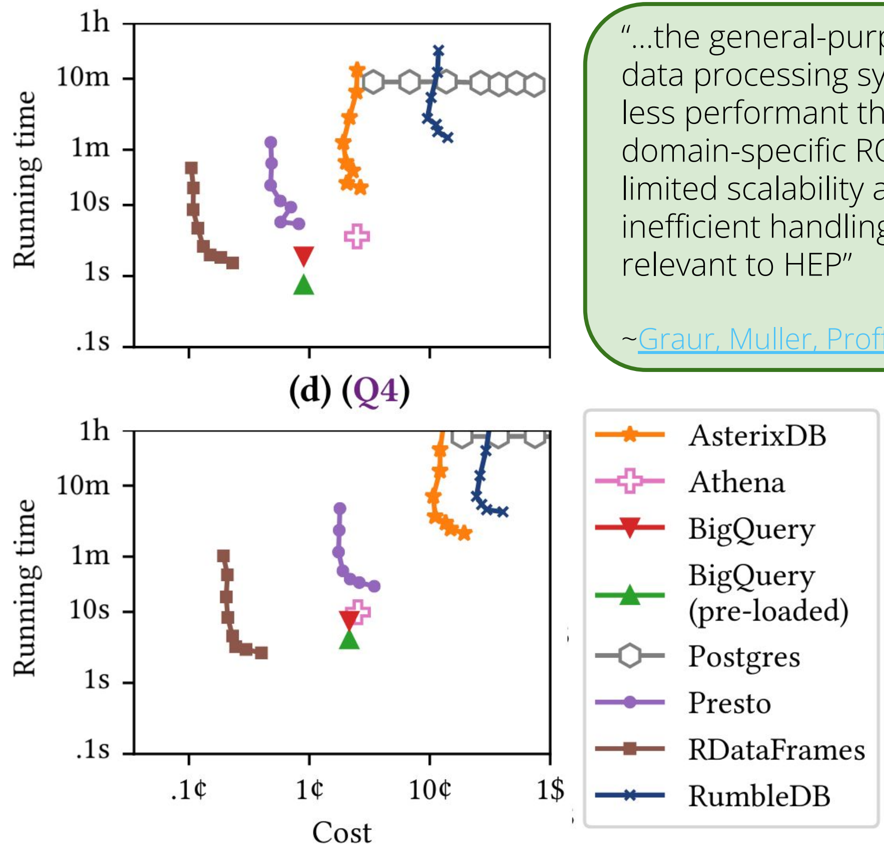
We care about performance. A lot.

Fu

Nano
when

zlib-c
128 t
~CMS

Hist Type
ROOT THnD
ROOT THnD
ROOT THnD
Boost (“sta”)
Boost (“sta”)
Boost (“sta”)
Boost (“sta”)



“...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 enables fast turnaround for complex analysis use cases
- RDataFrame scales well to many cores, many nodes, many histograms
- Performance is always ongoing work: we are constantly looking for feedback/use cases

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*!)

HEP-specific ergonomics

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

RVecs: working with collections

Select and fill: quick one-liner

```
h = df.Define("pt", "muon_pt[abs(muon_eta) < 2]").Histo1D("pt")
```

See also the [user guide]

RVecs: working with collections

Select and fill: quick one-liner

```
h = df.Define("pt", "muon_pt[abs(muon_eta) < 2]").Histo1D("pt")
```

Compiled C++

```
RVecD selectPt(RVecD &pt, RVecD &eta) {
  return pt[abs(eta) < 2];
}

auto h = df.Define("pt", selectPt,
                   {"muon_pt", "muon_eta"})
           .Histo1D<RVecD>("pt");
```

Python+Numba **Current R&D**

```
def select_pt(muon_pt, muon_eta):
  return muon_pt[np.abs(muon_eta) < 2]

h = df.Define("pt", select_pt).Histo1D("pt")
```

See docs about injecting Python into RDF in v6.26.

See also the user guide

Lightweight physics objects (R&D)

Select some muons, plot their inv. mass (now)

```
df.Define("m", "muon_pt > 20 && abs(muon_eta) < 2.7")
  .Define("invmass", "InvariantMass(muon_pt[m], muon_eta[m], muon_phi[m], muon_mass[m])")
  .Histo1D("invmass")
```

With automatic aggregation of muon_* into muons (coming soon)

Current R&D

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

On-the-fly systematic variations

In ROOT 6.26 (experimental)

Python

```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")
```

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")
```

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")
```

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")
```

obtain all variations

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")
```

proceed as usual, as if working with nominal values only

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

obtain all variations

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

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

obtain all variations

Variations automatically propagate to selections, derived quantities and results.

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

Only needed quantities are re-computed, all in **one event loop**.

RDF ⇔ NumPy arrays

v6.18

- **TTree → NumPy** via RDataFrame

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

- **NumPy → RDataFrame**

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

Work in progress: **RDF ⇔ Awkward arrays**, see github.com/awkward-1.0/issues/588

...and more

- transparent support for RNTuple, aka TTree 2.0 (faster, smaller) with no code changes
- machine learning inference as part of the event loop (see next talk about SOFIE)
- definition of per-sample quantities, e.g. varying histogram weights for data/MC
- support for TTree chains, friends, indexed friends, TEntryLists
- custom aggregations/results
- automatic cut-flow reports
- ...

Lazy action	Description
Aggregate()	Execute a user-defined accumulation operation on the processed column values.
Book()	Book execution of a custom action using a user-defined helper object.
Cache()	Cache column values in memory. Custom columns can be cached as well, filtered entries are not cached. Users can specify which columns to save (default is all).
Count()	Return the number of events processed. Useful e.g. to get a quick count of the number of events passing a Filter.
Display()	Provides a printable representation of the dataset contents. The method returns a **ROOT::RDF::RDisplay()** instance which can print a tabular representation of the data or return it as a string.
Fill()	Fill a user-defined object with the values of the specified columns, as if by calling `Obj.Fill(col1, col2, ...)`.
Graph()	Fills a **TGraph** with the two columns provided. If multi-threading is enabled, the order of the points may not be the one expected, it is therefore suggested to sort if before drawing.
GraphAsymmErrors()	Fills a **TGraphAsymmErrors**. If multi-threading is enabled, the order of the points may not be the one expected, it is therefore suggested to sort if before drawing.
Histo1D(), Histo2D(), Histo3D()	Fill a one-, two-, three-dimensional histogram with the processed column values.
HistoND()	Fill an N-dimensional histogram with the processed column values.
Max()	Return the maximum of processed column values. If the type of the column is inferred, the return type is `double`, the type of the column otherwise.
Mean()	Return the mean of processed column values.
Min()	Return the minimum of processed column values. If the type of the column is inferred, the return type is `double`, the type of the column otherwise.
Profile1D(), Profile2D()	Fill a one- or two-dimensional profile with the column values that passed all filters.
Reduce()	Reduce (e.g. sum, merge) entries using the function (lambda, functor...) passed as argument. The function must have signature `T(T,T)` where `T` is the type of the column. Return the final result of the reduction operation. An optional parameter allows initialization of the result object to non-default values.
Report()	Obtain statistics on how many entries have been accepted and rejected by the filters. See the section on **named filters** for a more detailed explanation. The method returns a **ROOT::RDF::RCutFlowReport** instance which can be queried programmatically to get information about the effects of the individual cuts.
Stats()	Return a **TStatistic** object filled with the input columns.
StdDev()	Return the unbiased standard deviation of the processed column values.
Sum()	Return the sum of the values in the column. If the type of the column is inferred, the return type is `double`, the type of the column otherwise.
Take()	Extract a column from the dataset as a collection of values, e.g. a `std::vector<float>` for a column of type `float`.

Instant action	Description
Foreach()	Execute a user-defined function on each entry. Users are responsible for the thread-safety of this callable when executing with implicit multi-threading enabled.
ForeachSlot()	Same as **Foreach()**, but the user-defined function must take an extra `unsigned int slot` as its first parameter. `slot` will take a different value, `0` to `nThreads - 1`, for each thread of execution. This is meant as a helper in writing thread-safe **Foreach()** actions when using **RDataFrame** after **ROOT::EnableImplicitMT()**. **ForeachSlot()** works just as well with single-thread execution: in that case `slot` will always be `0`.
Snapshot()	Write the processed dataset to disk, in a new **TTree** and **TFile**. Custom columns can be saved as well, filtered entries are not saved. Users can specify which columns to save (default is all). Snapshot, by default, overwrites the output file if it already exists. **Snapshot()** can be made *lazy* setting the appropriate flag in the snapshot options.

Queries

These operations do not modify the dataframe or book computations but simply return information on the **RDataFrame** object.

Operation	Description
Describe()	Get useful information describing the dataframe, e.g. columns and their types.
GetColumnNames()	Get the names of all the available columns of the dataset.
GetColumnType()	Return the type of a given column as a string.
GetColumnTypeNamesList()	Return the list of type names of columns in the dataset.
GetDefinedColumnNames()	Get the names of all the defined columns.
GetFilterNames()	Return the names of all filters in the computation graph.
GetNRuns()	Return the number of event loops run by this **RDataFrame** instance so far.
GetNSlots()	Return the number of processing slots that **RDataFrame** will use during the event loop (i.e. the concurrency level).
SaveGraph()	Store the computation graph of an **RDataFrame** in DOT format (graphviz) for easy inspection. See the **relevant section** for details.

The RDataFrame cheat sheet

Concluding remarks

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**.

With an ambitious plan of work, it is **critical to focus on the right features** - with your help!

Where to find us

Documentation

[RDF user guide]

[RDF tutorials]

[New ROOT manual]

User support
root-forum.cern.ch

Bug reports
github.com/root-project/root/issues

Development discussion
mattermost.web.cern.ch/root

Back-up

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

Side note: painless ROOT installation

```
$ yum install root
$ pacman -Syu root
$ brew install root
$ conda create -n cern-root -c conda-forge root
$ docker run -it rootproject/root
```

ROOT packages available upstream in **Fedora**, **Arch**, **Gentoo**, **CentOS** (via EPEL).

Conda, **Snap**, **Homebrew** & **Macports** packages also available (see root.cern/install).

Official **Docker** images at Dockerhub.

All of this only possible thanks to several amazing community members!

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

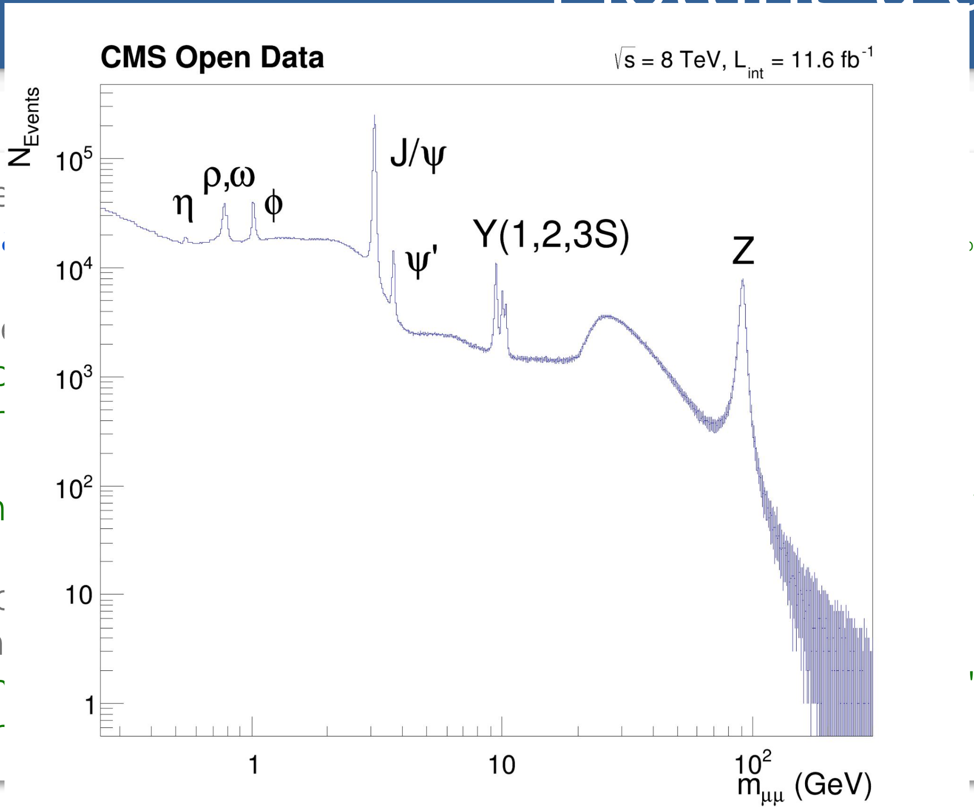
Python

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

# 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]

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.

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)")
  ...
```

See also the [user guide]

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.

Python functions in RDF with Numba

v6.24

Python

```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

```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.

Vary example: jet_E and MET

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.

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.

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

C++

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

the expression returns an array of values *per column*

- 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++

```cpp
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

```cpp
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.

Switch from TTree to RNTuple in RDF

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

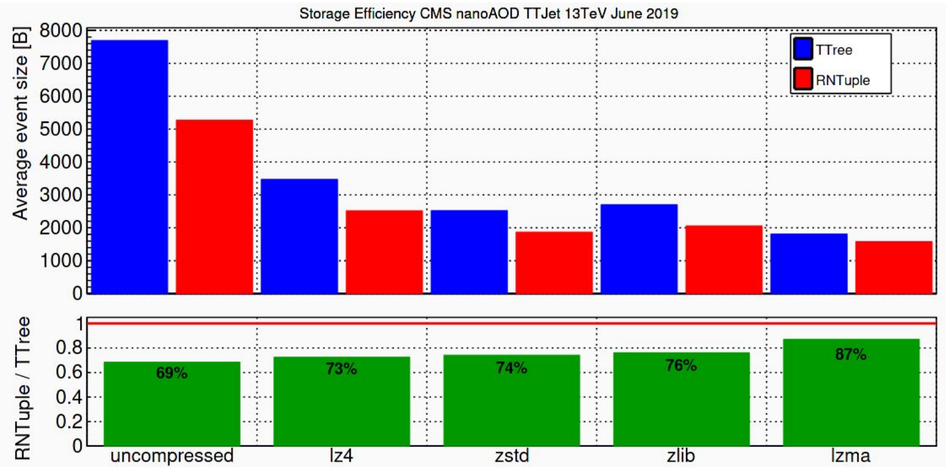
Run a multi-thread event loop

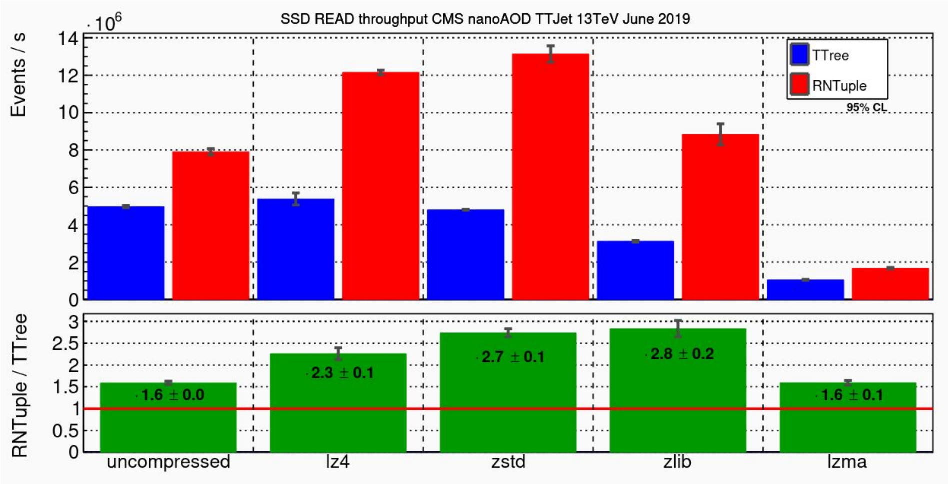
on this RNTuple

all other code stays the same

RDataFrame enables a **seamless transition to "TTree 2.0", RNTuple**.

RNTuple: smaller, faster than TTree



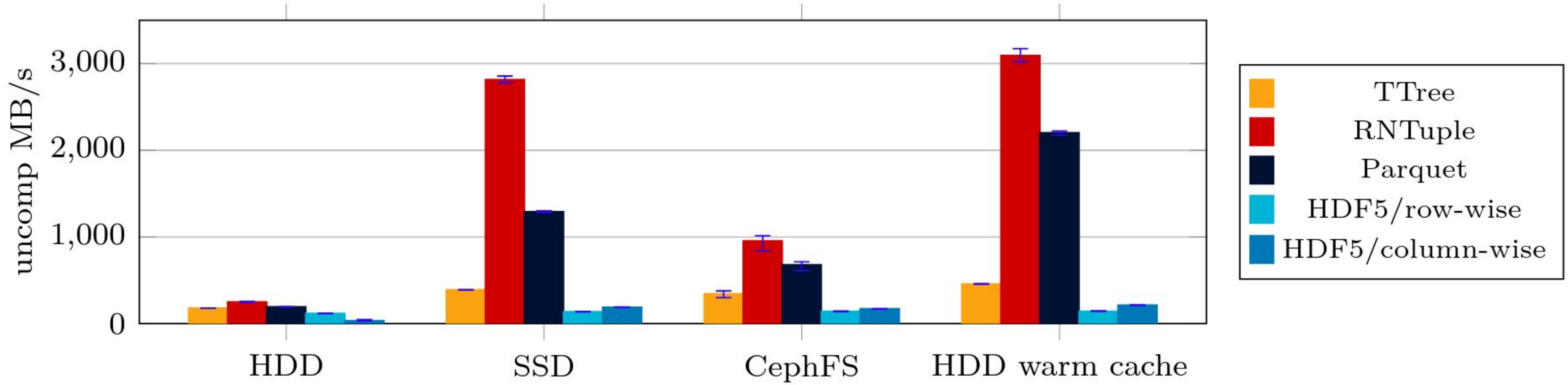


Goal

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

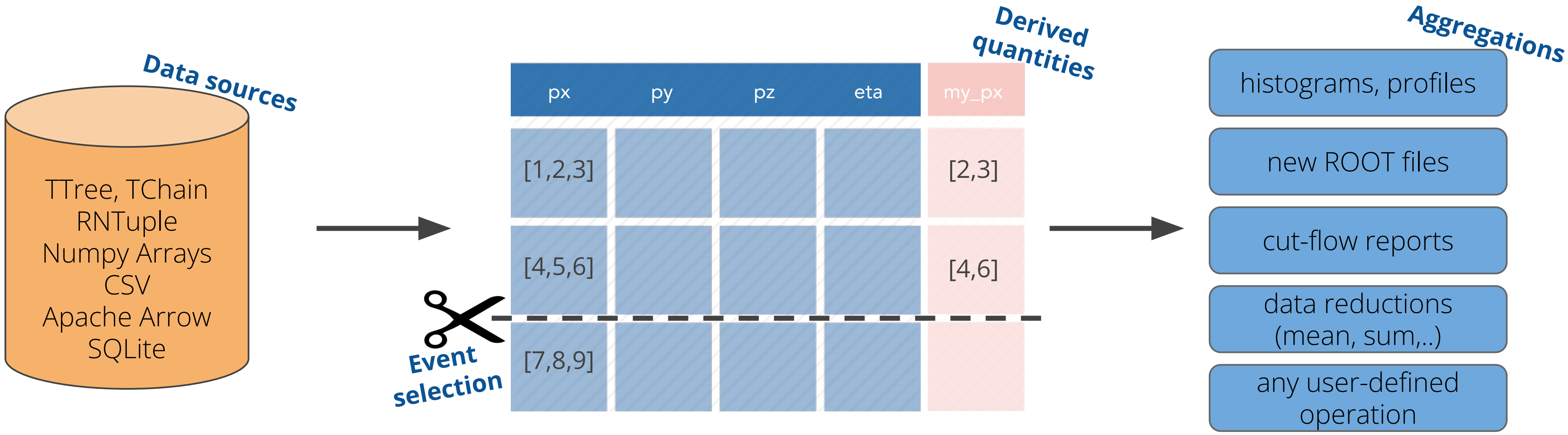
[Blomer et al., "Evolution of the ROOT Tree I/O", CHEP 2019]

RNTuple: the fastest HEP I/O

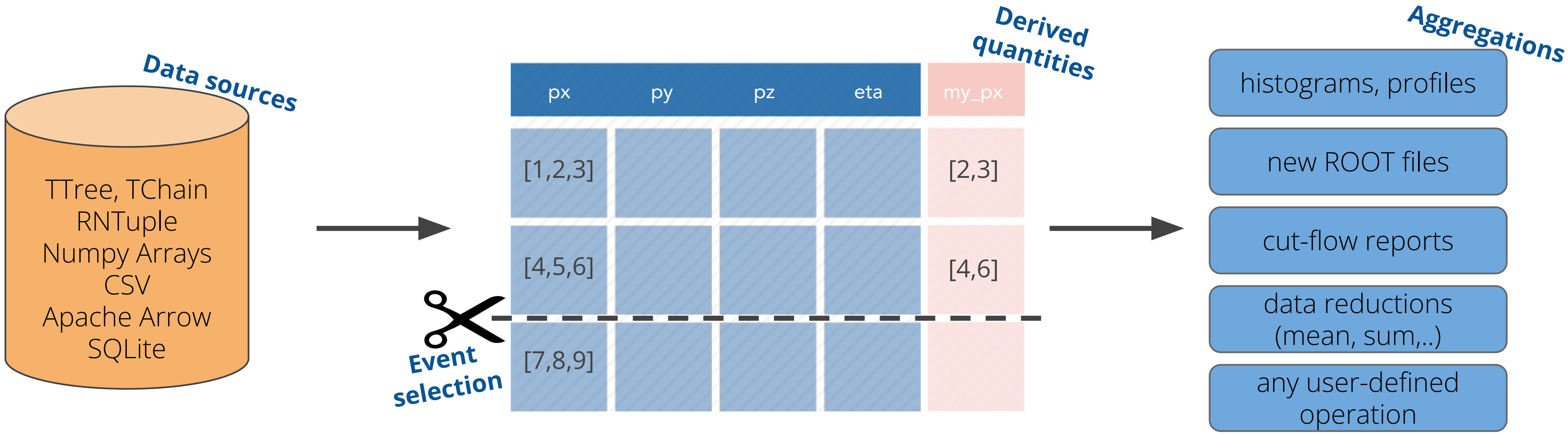


[J. Lopez Gomez, RNTuple: Performance, Updates and Outlook for 2022]

HEP data processing in a nutshell



HEP data processing in a nutshell



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.