Teaching RDataFrame

Enrico Guiraud for the ROOT team
ROOT train the trainers, 12/5/2022

This talk

As ROOT team, **we switched to teaching RDataFrame** as the one interface to interact with data.

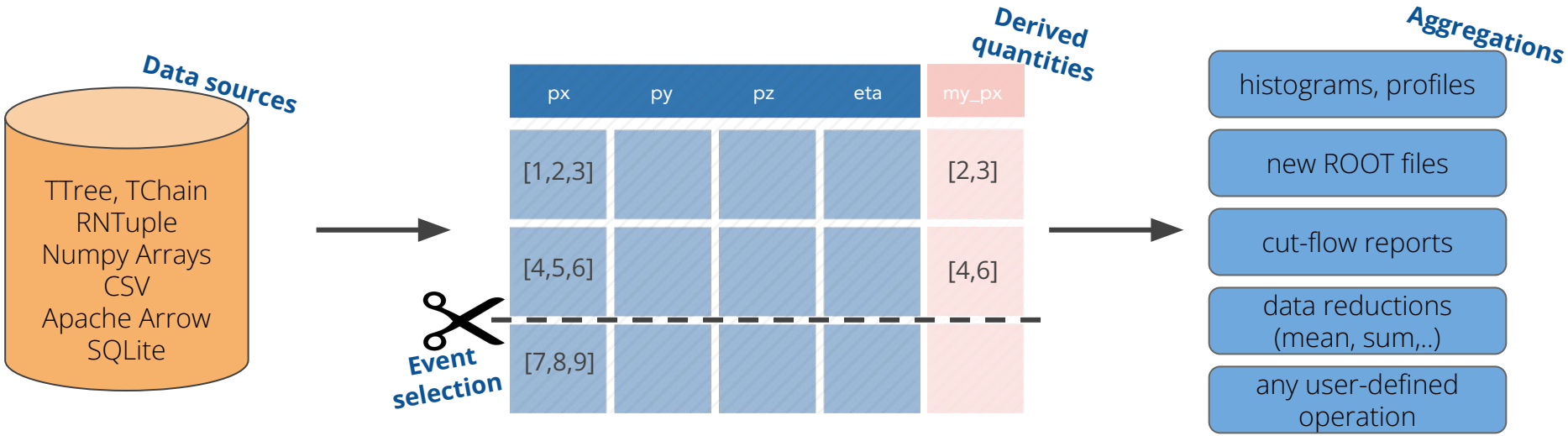
I'll provide **an overview of concepts**, programming model, useful features, common patterns and pitfalls in RDataFrame.

I will not provide more training material, these slides are meant for the trainers, not the trainees.

I will be referring to the state of RDF as of v6.26.

General concepts for trainers

The mental model



The programming model

```
ROOT::EnableImplicitMT();
ROOT::RDataFrame df("nMuons >= 2");
auto df2 = df.Filter("nMuons >= 2")
  .Define("good_pts", "muon_pt[abs(muon_eta) < 2]");
RResultPtr<TH1D> h = df2.Histo1D("good_pts");
h->Draw();
```

event selection

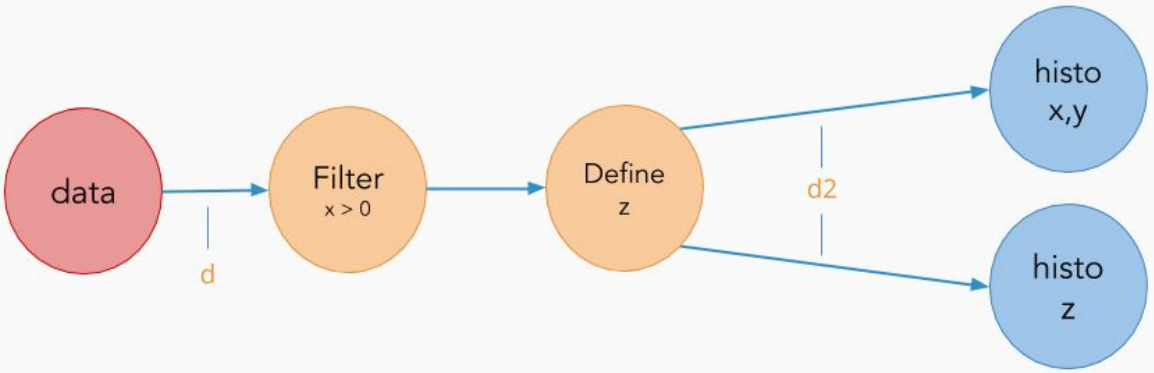
derived quantities, object selections

aggregations

event loop runs here, produces all results in one event loop

At each step **users can inject custom code**
(string expressions or actual C++ callables – lambdas, functions, functors).
The **injection can happen from Python** too, thanks to PyROOT.

Internally, a lazy computation graph




```
// d2 is a new data-frame, a transformed version of d
auto d2 = d.Filter("x > 0")
           .Define("z","x*x + y*y");
// make histograms out of it
auto hz = d2.Histo1D("z");
auto hxy = d2.Histo2D({"hxy","hxy",16,-1,1,64,-1,1},"x","y");
```

RDataFrame is a swiss-army knife

A single tool to cover most use cases

- inspecting a dataset (Display, Describe, GetColumnNames, GetColumnType)
- quick exploration (Range, Report)
- generating new datasets (RDataFrame(100).Define("pythiaOutput", ...).Snapshot(...))
- ntuple -> ntuple transformation (Alias, Redefine, Snapshot)
- ROOT data -> NumPy arrays (AsNumpy)
- complex analysis use cases (indexed friend trees, Vary, DefinePerSample, Book)
- scaling out to clusters (distributed execution, PROOF successor)

Debug features

- SaveGraph
- verbose mode with event loop timing

Resources

On top of general ROOT material:

- RDF user guide: includes a cheat-sheet, a crash course as well as in-depth explanations
- RDF tutorials: in C++ and Python, include examples of simple analyses
- opendata benchmarks: example implementation of 8 typical queries

RDataFrame basics for novice users

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

Computations are run lazily

```
df = ROOT.RDataFrame("Events", "root://…/somefile.root")
df = df.Filter("nMuons == 2")
```

construct RDF object,
apply an event-level selection

```
h1 = df.Histo1D("muon_pt")
h2 = df.Histo1D("muon_pt", "w")
```

h1 and h2 are result *proxies*
(empty for now)

```
h1.Draw()
```

event loop runs here, upon *first access to any result*
(and both histograms are filled)

Lazy execution makes it possible to book complex analysis workflows
and then produce all results in a single event loop.

Booking before executing (Python)

```
df = ROOT.RDataFrame("Events", "root://…/somefile.root")
df = df.Filter("nMuons == 2")

histos = []
for _ in range(1000):
  h = df.Define(...).Histo1D(...)
  histos.append(h)

histos[0].GetValue()
```

construct RDF object,
apply an event-level selection

event loop runs here, upon *first access to any result*
(and all histograms are filled)

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)")
  …
```

Quite verbose, so work in progress on supporting collection aggregations (muon_* -> muons).

RDF ⇔ NumPy arrays

- **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.

Distributed RDataFrame (Python-only)

Local

```
from ROOT import RDataFrame
```

```
df = RDataFrame('treename', 'filename.root')
```

Distributed (dask)

```
import ROOT
RDataFrame = ROOT.RDF.Experimental.Distributed.Dask.RDataFrame
```

```
from dask.distributed import Client
client = Client(HTCondorCluster(...))
df = RDataFrame('treename', 'filename.root', daskclient=client)
```

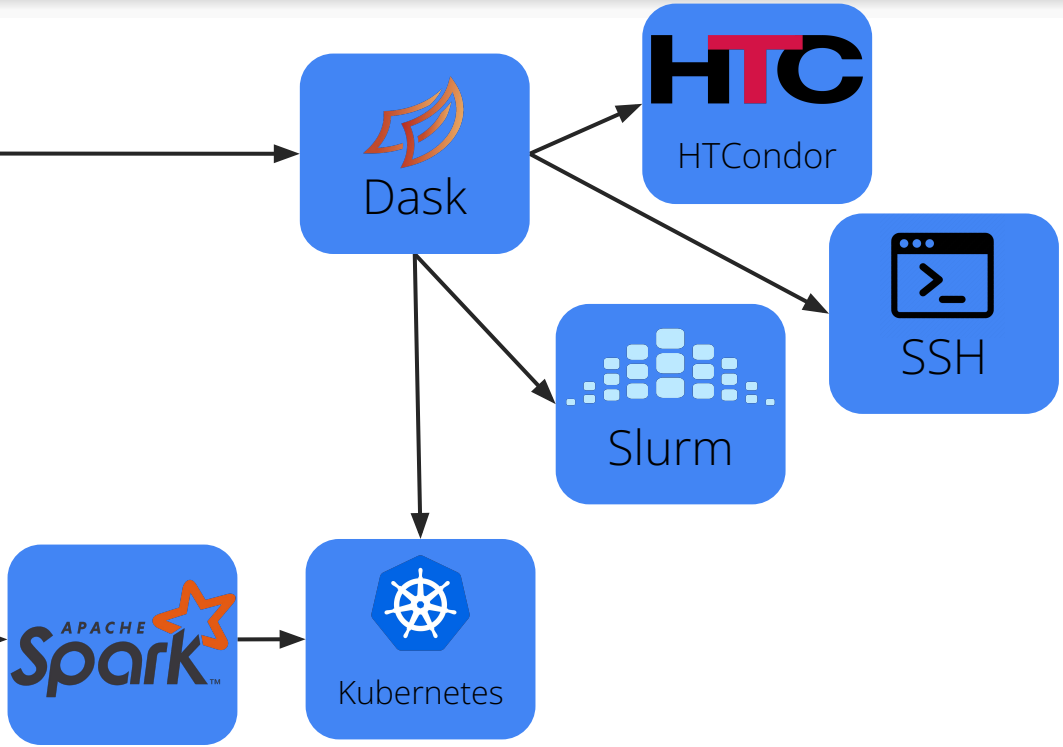
```
df2 = df.Filter(...).Define(...)
h1 = df2.Histo1D(...)
h1.Draw()
```

See Vincenzo's talk and the docs for more on distributed RDF.

Many supported backends

```
from dask.distributed import Client
from dask_jobqueue import HTCondorCluster
client = Client(HTCondorCluster(n_workers=64))
df = RDataFrame('tree', 'dir/*.root', daskclient=client)
```

```
from pyspark import SparkContext
sc = SparkContext('spark://IP:PORT')
df = RDataFrame('tree', 'dir/*.root', sparkcontext=sc)
```



Demo notebooks with Spark and Dask

Common pitfalls

Multithreading

- **Snapshot** writes entries out of order w.r.t. input
- Column **rdfentry_** does not correspond to input TChain global entry number

Python usage

- a few RDF methods are not supported in distributed RDF (but work in local Python)
- Numba-jitted functions that call NumPy methods have terrible scaling
- grabbing variables from Python:

```
cut = 42
df2 = df.Filter(f"x > {cut}")
```

```
cut = 42
df2 = df.Filter(ROOT.HelperCppFunctor(cut), ["x"])
```

Level 2, for more advanced users

Modularity comes in two flavors

Factoring out RDF manipulation

Python

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

df = apply_cuts(df)
```

C++

```cpp
RDF::RNode ApplyCuts(RDF::RNode df) {
  auto df2 = df.Filter(...).Filter(...);
  return df2;
}

auto df2 = ApplyCuts(df);
```

Factoring out user-defined logic

Python

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

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

C++

```cpp
class Evaluator {
  RVecD operator()(RVecD& x, RVecD& y) { … }
  static const vector<string> inputs = {"x", "y"};
};

auto df2 = df.Define("x", Evaluator{}, Evaluator::inputs);
```

On-the-fly systematic variations

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.
Multithread and **distributed** execution **just works**.

On-the-fly systematic variations

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")
```

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

On-the-fly systematic variations

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

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

On-the-fly systematic variations

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.
Multithread and **distributed** execution **just works**.

On-the-fly systematic variations

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 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.
Multithread and **distributed** execution **just works**.

Python functions in RDF with Numba

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()
```

As a general rule with numba, but especially when injecting numba-jitted functions in RDataFrame's event loop, **prefer explicit for loops to NumPy functions** for performance.

Definition of per-sample values

```cpp
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.

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.

Summary

Summary

RDataFrame is a powerful interface that covers use cases from simple data explorations, to dataset generation or transformation, to complex analyses.

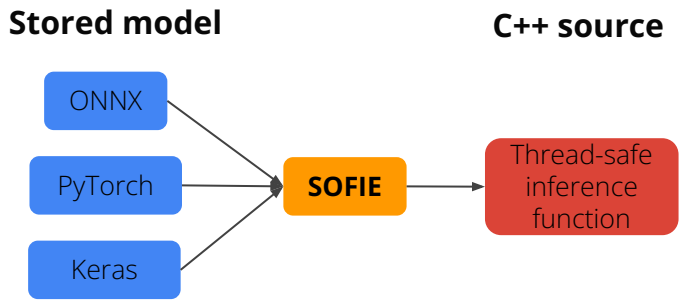
The main conceptual hurdles when first approaching RDataFrame are typically not being directly in control of the event loop and when it runs, working with collections and some quirks in Python usage.

Let us know what your experience teaching RDF is!

EOF

Back-up content

ML inference in RDF with SOFIE




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

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

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

Also see this talk by Lorenzo on Wednesday.

Vary: 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")
```

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

Stateful, thread-aware Filters/Defines

Example code here.

Varying multiple columns together

```cpp
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>`

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.

Varying multiple columns together

the expression returns an array of values *per column*

```cpp
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.