Introduction to RDataFrames

And a short peek into uproot

`ROOT.RDataFrame` is ROOT's modern interface to data analysis

- easy to use
- scalable
- interoperable with the Python ecosystem

RDataFrames - basic functionality

```
import ROOT
```

The `RDataFrame` constructor takes the name of a tree and one or more files.

```
df = ROOT.RDataFrame("treename", "file.root")
# or
df = ROOT.RDataFrame("treename", ["file1.root", "file2.root", ...])
```

The location of the files can be local or remote.

In this example, we're loading data from the ATLAS Opendata set.

```
baseurl = ("https://atlas-opendata.web.cern.ch/"
    "atlas-opendata/samples/2020/2lep/Data"
)
fileurls = [f"{baseurl}/data_{i}.2lep.root" for i in "ABCD"]

df = ROOT.RDataFrame("mini", fileurls)
```

Inspect the contents of a tree

```
df.Describe()   # new in ROOT v6.26.00
```

```
Dataframe from TChain mini in files
  https://atlas-opendata.web.cern.ch/atlas-opendata/samples/2020/2lep/Data/data_A.2lep.root
  https://atlas-opendata.web.cern.ch/atlas-opendata/samples/2020/2lep/Data/data_B.2lep.root
  https://atlas-opendata.web.cern.ch/atlas-opendata/samples/2020/2lep/Data/data_C.2lep.root
  https://atlas-opendata.web.cern.ch/atlas-opendata/samples/2020/2lep/Data/data_D.2lep.root

Property                Value
--------                -----
Columns in total           81
Columns from defines        0
Event loops run             0
Processing slots            1

Column                          Type                                Origin
------                          ----                                ------
SumWeights                      Float_t                             Dataset
XSection                        Float_t                             Dataset
channelNumber                   Int_t                               Dataset
ditau_m                         Float_t                             Dataset
eventNumber                     Int_t                               Dataset
jet_E                           ROOT::VecOps::RVec<float>           Dataset
```

Inspect the contents of one or more columns

```
df.Display(["lep_E", "lep_pt"], 5).Print()
```

```
+-----+----------+----------+
| Row | lep_E    | lep_pt   |
+-----+----------+----------+
| 0   | 147913.f | 33144.9f |
|     | 21271.8f | 21067.4f |
+-----+----------+----------+
| 1   | 42967.8f | 42871.2f |
|     | 55135.4f | 37205.4f |
+-----+----------+----------+
| 2   | 35314.3f | 33070.2f |
|     | 126647.f | 29222.1f |
...
```

Functionality for data analysis

Think of an RDataFrame as a table that you can use to compute new columns from existing ones and filter based on various conditions.

```
df_filtered = df.Filter("condition", "optional name for this cut")
```

The condition can be passed either as a C++ expression in a string or as a python function.
Defining new columns works in the same way:

```
df_new = df_filtered.Define("columnname", "c++ expression")
```

important note: `Filter` and `Define` do not mutate the dataframe object but rather return a new object. These operations are also **lazy** meaning that nothing is computed until the result is actually requested by the user.

Experimental new feature: Systematic variations

RDataFrames offer a declarative way to define systematic variations of columns:

```
nominal_df = df.Vary("pt", "ROOT::RVecD{pt*0.9, pt*1.1}", ["down", "up"])
                .Define(...)
                .Filter(...)
histo = ROOT.RDF.Experimental.VariationsFor(nominal_df)
histo["nominal"].Draw()
histo["pt:down"].Draw("SAME")
```

Example

In this example, we're trying to plot the invariant mass of all $e^+e^-e^+e^-$, $\mu^+\mu^-\mu^+\mu^-$ and $e^+e^-\mu^+\mu^-$ pairs.

First, filter for exacly 4 leptons with total 0 charge and the correct lepton flavours:

```
%%time
df_filtered = (df
               .Filter("lep_n == 4", "exacly four leptons")
               .Filter("lep_charge[0] + lep_charge[1] + lep_charge[2] + lep_charge[3] == 0",
                       "zero charge")
               .Define("sum_lep_type",
                       "lep_type[0] + lep_type[1] + lep_type[2] + lep_type[3]")
               .Filter("sum_lep_type == 44 || sum_lep_type == 48 || sum_lep_type == 52",
                       "lepton types")
)
```

```
CPU times: user 418 µs, sys: 61 µs, total: 479 µs
Wall time: 505 µs
```

Because of RDataFrame's lazyness, this operation returns almost instantly. The computations are only "booked".

now we define a new column for the invariant mass of the four leptons:

```
%%time
df_new = df_filtered.Define("mllll",
        """(ROOT::Math::PtEtaPhiEVector(lep_pt[0], lep_eta[0], lep_phi[0], lep_E[0])
           + ROOT::Math::PtEtaPhiEVector(lep_pt[1], lep_eta[1], lep_phi[1], lep_E[1])
           + ROOT::Math::PtEtaPhiEVector(lep_pt[2], lep_eta[2], lep_phi[2], lep_E[2])
           + ROOT::Math::PtEtaPhiEVector(lep_pt[3], lep_eta[3], lep_phi[3], lep_E[3])
        ).M() * 0.001"""
)
```

```
CPU times: user 13.4 ms, sys: 1.6 ms, total: 15 ms
Wall time: 18.7 ms
```

again, we can observe the lazyness... nothing is actually computed yet!

Interoperability

The columns in RDataFrames can be converted to numpy arrays for use cases where you don't want to continue working with ROOT.
Converting to numpy is one example of the user requesting to get the data and therefore triggering the execution of all previously booked computations:

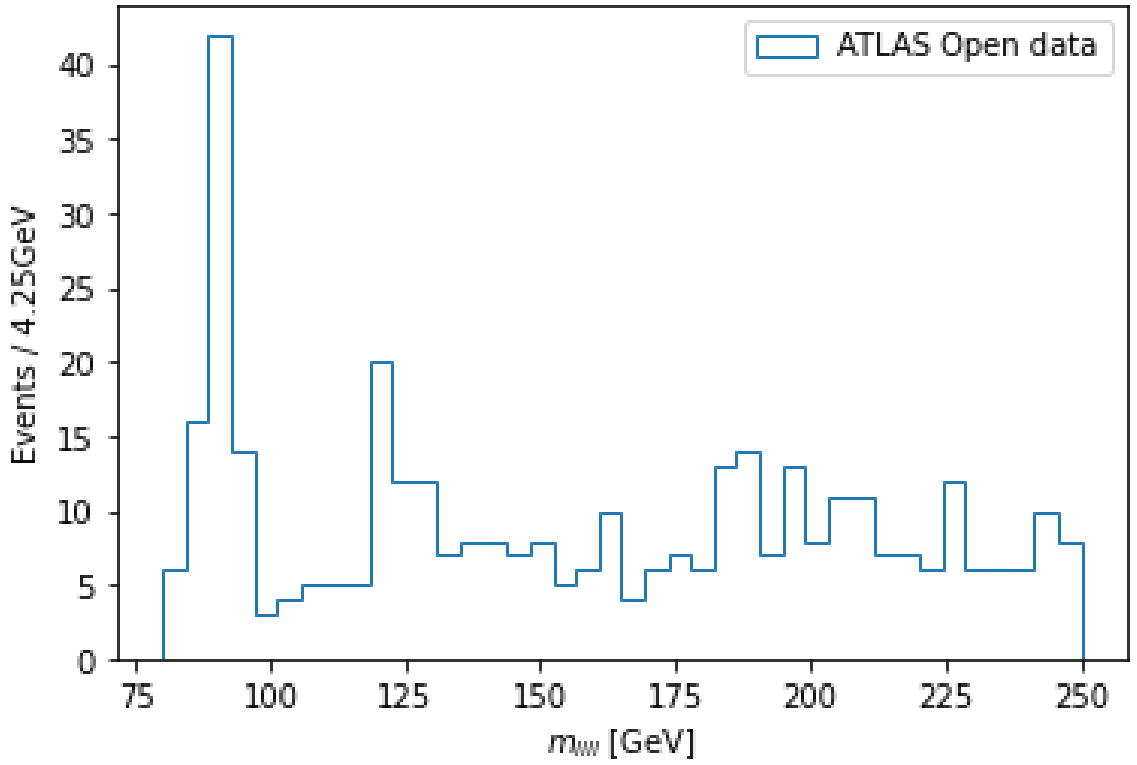
```
%%time
mllll = df_new.AsNumpy(["mllll"])
```

```
CPU times: user 16.4 s, sys: 5.08 s, total: 21.5 s
Wall time: 2min 25s
```

We get back a dict

```
{'mllll': ndarray([649.27209146,
87.90711324, ..., 305.91470474])}
```

```
import matplotlib.pyplot as plt
plt.hist(
    mllll["mllll"],
    range=(80, 250),
    bins=40,
    histtype="step",
    label="ATLAS Open data"
)
plt.xlabel("$m_{llll}$ [GeV]")
plt.ylabel("Events / 4.25GeV")
plt.legend()
plt.show()
```



Inspection

RDataFrames offer easily accessible methods to track down what actually happened in a computation.

For example get a report of the efficiencies of each filter applied:

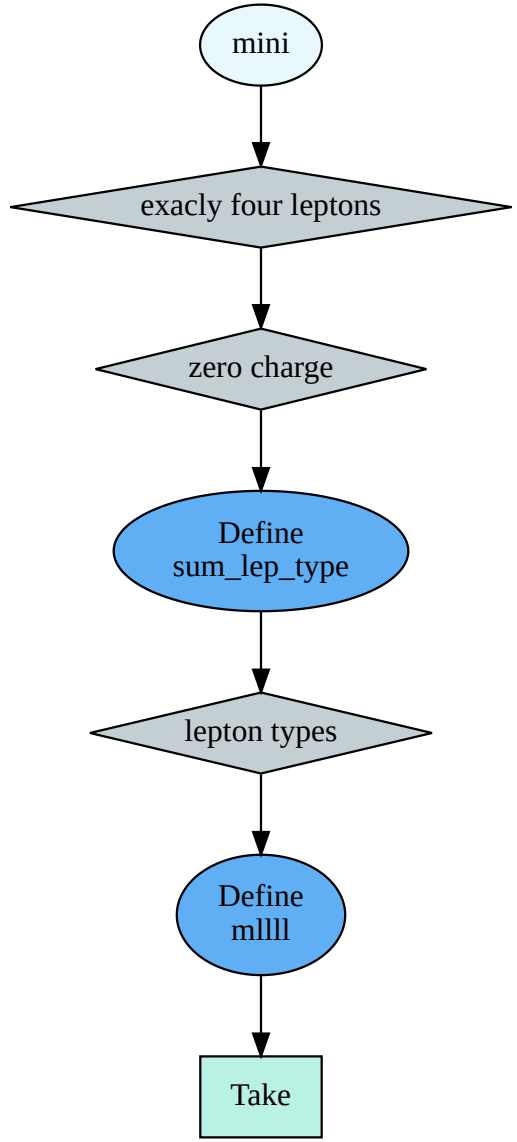
```
df.Report().Print()
```

```
exacly four leptons: pass=827        all=12205790   -- eff=0.01 % cumulative eff=0.01 %
zero charge: pass=644        all=827        -- eff=77.87 % cumulative eff=0.01 %
lepton types: pass=506        all=644        -- eff=78.57 % cumulative eff=0.00 %
```

Or get the computational graph

```
# visualize the computation graph
ROOT.RDF.SaveGraph(df, "DAG.dot")

from graphviz import Source
Source.from_file("DAG.dot")
```

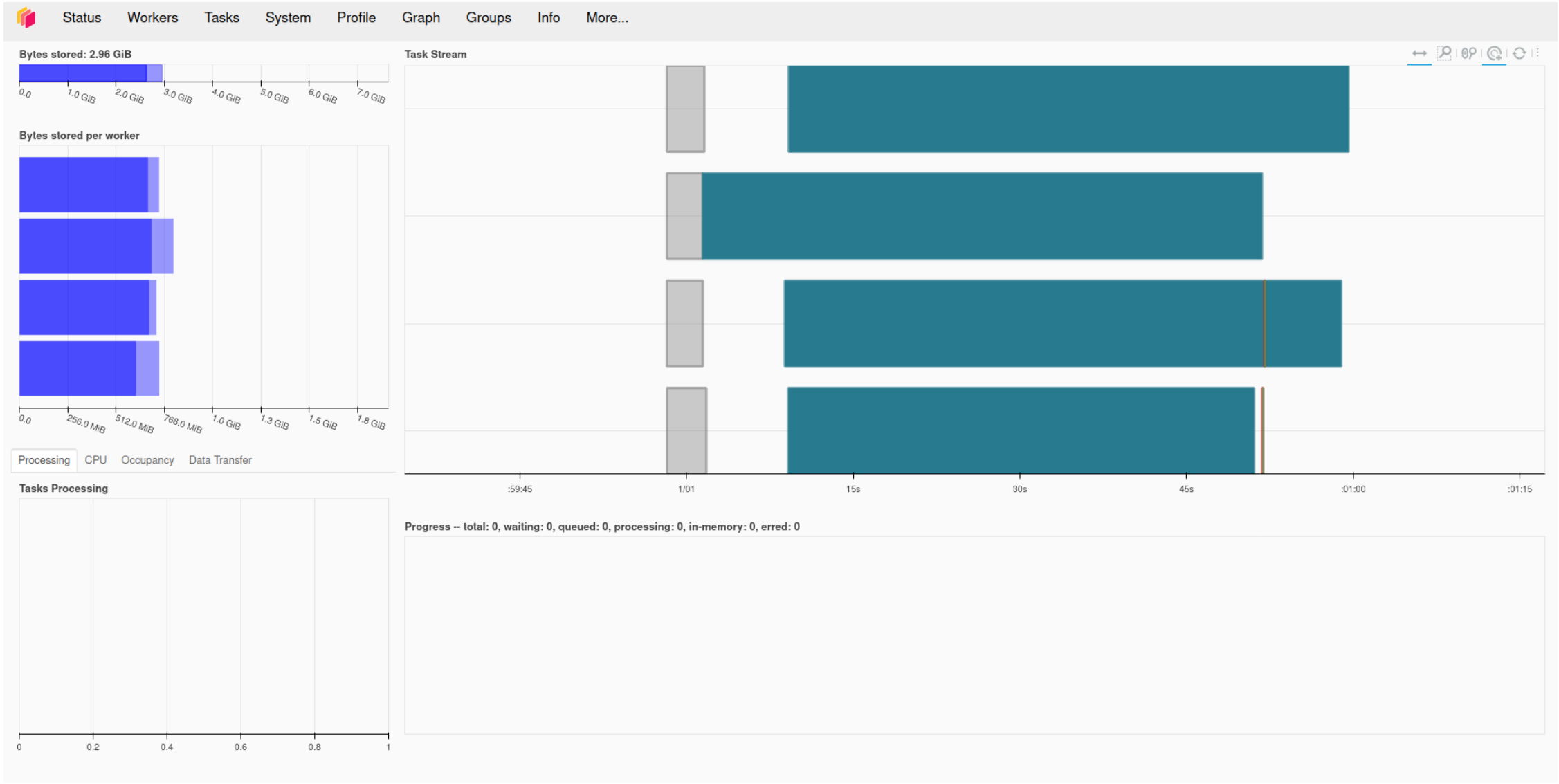


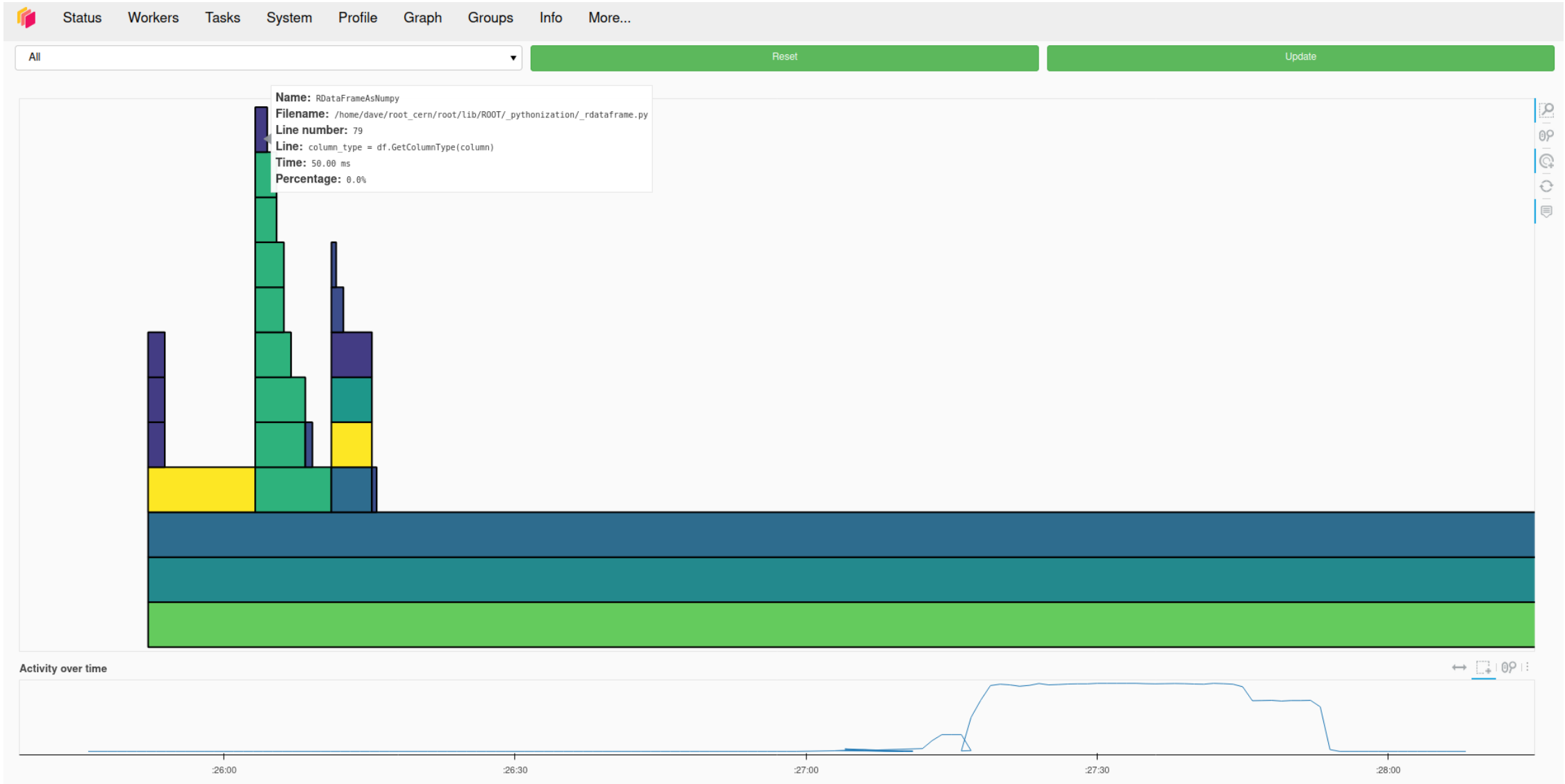
Scaling up

RDataFrames have the (as of now still experimental) option to run distributed on a cluster (eg. Dask) to scale up your analysis.

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

df = DistRDataFrame("treename", filelist, daskclient=my_client)
```


Status
Workers
Tasks
System
Profile
Graph
Groups
Info
More...
Bytes stored: 2.96 GiB
0.0
1.0 GiB
2.0 GiB
3.0 GiB
4.0 GiB
5.0 GiB
6.0 GiB
7.0 GiB
Bytes stored per worker
0.0
256.0 MiB
512.0 MiB
768.0 MiB
1.0 GiB
1.3 GiB
1.5 GiB
1.8 GiB
Processing
CPU
Occupancy
Data Transfer
Tasks Processing
0
0.2
0.4
0.6
0.8
1
Task Stream
:59:45
1/01
15s
30s
45s
:01:00
:01:15
Progress -- total: 0, waiting: 0, queued: 0, processing: 0, in-memory: 0, erred: 0


Status
Workers
Tasks
System
Profile
Graph
Groups
Info
More...
All
Reset
Update
Name: RDataFrameAsNumpy
Filename: /home/dave/root_cern/root/lib/ROOT/_pythonization/_rdataframe.py
Line number: 79
Line: column_type = df.GetColumnType(column)
Time: 50.00 ms
Percentage: 0.0%
Activity over time
:26:00
:26:30
:27:00
:27:30
:28:00

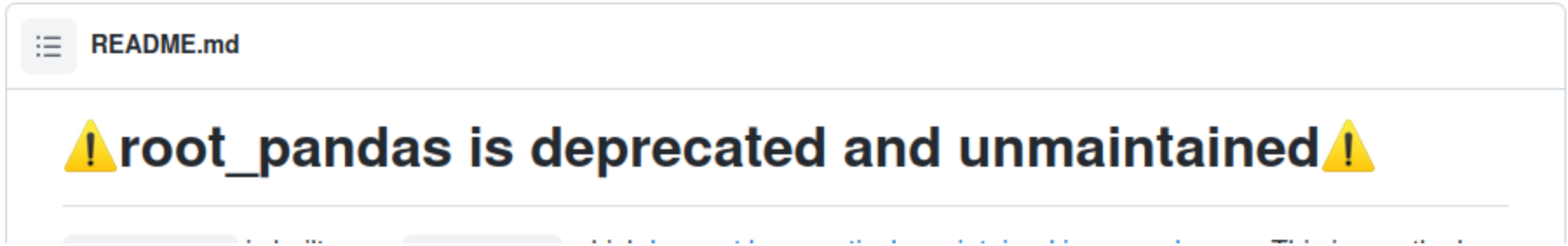
Still not convinced?

Rather use pure python?

Do it right!

uproot

(Stop using `root-pandas`!)



uproot is a pure python package for root I/O

```
import uproot

with uproot.open("filepath.root:treename") as tree:
    for data in tree.iterate(["column1", "column2", ...], library="pd"):
        ...
```

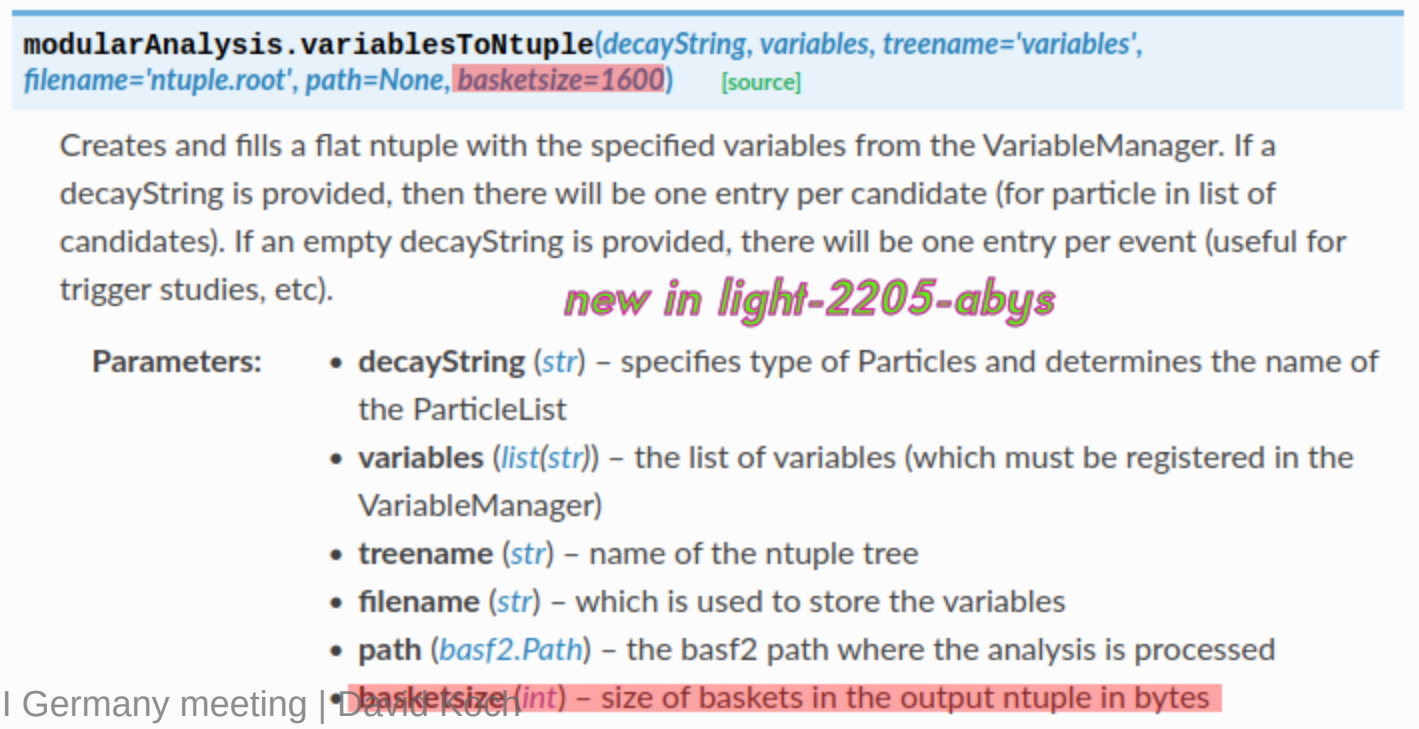
uproot can interpret data from root files as

- numpy arrays
- awkward arrays
- pandas dataframes

Short note about speed

If you have a workflow that uses basf2's `modularAnalysis.variablesToNtuples()` to generate root files and then continue to read them
→ you might find that uproot is slower than root-pandas; this is probably due to small basket sizes in the input file

set this parameter to a large number (eg. 20000000) for fast reading with uproot 🚙



modularAnalysis.variablesToNtuple(*decayString, variables, treename='variables', filename='ntuple.root', path=None, basketsize=1600*) [source]

Creates and fills a flat ntuple with the specified variables from the VariableManager. If a decayString is provided, then there will be one entry per candidate (for particle in list of candidates). If an empty decayString is provided, there will be one entry per event (useful for trigger studies, etc).

new in light-2205-abys

Parameters:

- **decayString** (*str*) – specifies type of Particles and determines the name of the ParticleList
- **variables** (*list(str)*) – the list of variables (which must be registered in the VariableManager)
- **treename** (*str*) – name of the ntuple tree
- **filename** (*str*) – which is used to store the variables
- **path** (*basf2.Path*) – the basf2 path where the analysis is processed
- **basketsize** (*int*) – size of baskets in the output ntuple in bytes