Data Analysis with ROOT

From data to results: Expressive, Easy and Fast

Lorenzo Moneta (CERN) for the ROOT team, E. Manca (SNS & CERN)
ACAT 2019, Saas-Fee, Switzerland
ROOT’s declarative analysis
Interoperability with the Python ecosystem
Real life examples
  - Event generator study
  - CMS W mass analysis and H→μμ study with systematics variations
  - Totem full analysis distributed with Apache Spark
Conclusions and plans

Unless explicitly stated, we refer to the ROOT 6.16 release
Declarative Analysis: RDataFrame
A recipe for efficient HEP analyses

➔ strive for a **simple programming model**
➔ expose modern, elegant interfaces that are **easy to use correctly** and hard to use incorrectly
➔ allow to **transparently benefit from parallelism**
ROOT Declarative Analysis: RDataFrame

Goals:
→ Be the **fastest** way to manipulate HEP data
→ Be the **go-to** ROOT analysis interface from laptop to cluster
→ Consistent interfaces in **Python and C++**
→ Top notch [documentation and examples](#)

Datasource
- ROOT CSV
- Apache Arrow
- ATLAS’ xAOD
- LHCb’s MDF
- <Your Format>

Customisation point, public interface!
An ergonomic, fast C++ dataframe

```cpp
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
```

Lazy execution one event loop
An ergonomic, fast C++ dataframe

```cpp
ROOT::EnableImplicitMT();                       // Run a parallel analysis
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
```

Lazy execution guarantees that all operations are performed in one event loop
ROOT::RDataFrame \texttt{df(dataset)};

\texttt{auto df2 = df.Filter("x > 0")}
\texttt{.Define("r2", "x*x + y*y");}

\texttt{auto rHist = df2.Histo1D("r2");}

\texttt{df2.Snapshot("newtree", "newfile.root");}

Write datasets to disk, also in parallel.
No templates: C++ → JIT → Python

C++

d.Filter[](double t) { return t > 0.; },{"theta"})
.Snapshot<vector<float>>("mytree","f.root","pt_x");

C++ with cling’s just-in-time compilation

d.Filter("theta > 0").Snapshot("mytree","f.root","pt_x");

PyROOT, automatically generated Python bindings

d.Filter("theta > 0").Snapshot("mytree","f.root","pt_x")
auto inMemDF = d.Filter("All(muon_eta < 2.5)")
  .Cache({"muon_eta"});

RVec reference guide (top notch doc, too!)
# Run input pipeline with C++ performance that can process TBs of data, reads from remote, ...

```python
import ROOT
df = ROOT.RDataFrame("tree", "file.root")
    .Filter("All(pt>30)", "Trigger requirement")
    .Filter("All(tight_iso)", "Quality cut")
    .Define("r", "sqrt(eta*eta + phi*phi)"")
```

# Extract selection w/ defined variables as numpy arrays
```
col_dict = df.AsNumpy(["r", "eta", "phi"])
```

# Wrap data with pandas
```
import pandas
p = pandas.DataFrame(col_dict)
print(p)
```

```
r     eta   phi
0  0.26   0.1 -0.5
1  1.0   -1.0  0.0
2  4.45   2.1  0.2
...```

All the power of RDF + possibility to convert to NumPy: coming in 6.16/02

See A More Pythonic, Interoperable and Modern PyROOT, 11/3 16:10 Steinmatte
import ROOT
import numpy

# Create an RDataFrame from a ROOT file
df = ROOT.RDataFrame("tree", "file.root")

# Declare Python callable to be visible from C++
@ROOT.DeclareCppCallable(["float", "float"], "float")
def func(x, y):
    return numpy.sqrt(x**2 + y**2)

# Call Python function from C++, e.g.,
# to define a new column in the RDataFrame
df2 = df.Define("r", "ROOT::func(eta, phi)")

- Compilation with Numba also possible
- Functionality available, focusing on the interfaces and programming model
import ROOT
import numpy

# Assume data represented by numpy arrays
x = numpy.array([[ ... ]])
y = numpy.array([[ ... ]])

# Construct an RDataFrame reading from the numpy arrays
df = ROOT.MakeNumpyDataFrame({'x': x, 'y': y})

# Perform transformations and actions on the data
df2 = df.Define('z', 'sqrt(x*x + y*y)')

A factory function returning a RDataFrame
Real Life Examples
Realistic analysis, 100 systematics

- 3400 nodes in the computation graph, heavy usage of RVec
- 1GB input file, NanoAOD format, LZMA compressed
- Reading+Decompressing: ~20% of the sequential runtime

Intel Core i7 7820X (8*2 cores, 3.60GHz)

Realistic Analysis, Large Computation Graph: Good Performance & Efficient Scaling
Does All This Scale?

E. Manca (SNS & CERN)
CMS W Mass Analysis
(with IO)

~ 1.5 MHz @ 90 Cores!

RDataFrame Scales on Many Cores
Does All This Scale?

RDataFrame Scales on Many Cores

E. Manca (SNS & CERN)
CMS W Mass Analysis (with IO)

~ 1.5 MHz @ 90 Cores!

Under Investigation!

192*2 cores Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz

48 and 64 physical cores

X. Valls (CERN TechLab)
Gen. Level study (no IO)

Linear scaling on physical cores

Events/s
Investigate and prototype a complement to PROOF

- Parallelism on many nodes
- Transparent distribution
- Support several different backends

```python
d = RDataFrame ("t", dataset)
f = d.Define(...)
  .Define(...)
  .Filter(...)

h1 = f.Histo1D(...)  
h2 = f.Histo1D(...)  
h3 = f.Histo1D(...)```

Not in 6.16
Working prototype available!

JavierCVilla/PyRDF

Re-use RDF interface: Minimal/No change in analysis code
import ROOT

# Initialize RDataFrame object
df = ROOT.ROOT.RDataFrame(dataset)

# Define operations
df2 = df.Filter("x > 0")
    .Define("r2", "x*x + y*y")
rHist = df2.Histo1D("r2")

# Display histogram
rHist.Draw()

import PyRDF

# Initialize RDataFrame object
df = PyRDF.RDataFrame(dataset)

# Define operations
df2 = df.Filter("x > 0")
    .Define("r2", "x*x + y*y")
rHist = df2.Histo1D("r2")

# Display histogram
rHist.Draw()
RDF+Spark Scaling

- Revisited published TOTEM analysis
- CS thesis about this effort
- It works and there is room for further improvement
Wrap-up
ROOT offers a production grade declarative analysis interface

- Easy, fast, scalable: demonstrated with large real life use cases
- Interoperable with Python
- Top notch documentation and examples

Bright future ahead:

- Further develop the distributed analysis demonstrator
- Transform today’s use-case in a long-running benchmark suite
- Put in production PyROOT related developments
- Make RDF the data reading backend for machine learning