Declarative Analysis in ROOT with RDataFrame

S. Wunsch for the ROOT team
HSF Workshop Jefferson Lab
ROOT’s declarative analysis

Array syntax

Real life examples
- CMS W mass analysis and H→μμ study with systematics variations
- Totem full analysis distributed with Apache Spark

Keywords, actions and transformations

Interoperability with Python

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 **transparency benefit from parallelism**
ROOT Declarative Analysis: RDataFrame

- **Datasource**
  - ROOT CSV
  - Apache Arrow
  - [ATLAS’ xAOD]
  - [LHCb’s MDF]
  - [Your Format]

- **Define**
  - pt
  - eta
  - phi
  - mass
  - myvar

- **Range Filter**

- **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](#)
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
An ergonomic, fast C++ dataframe

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**
Analyses as computation graphs

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

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

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");
Examples from the tutorials

- Fully runnable examples with data and code
- More realistic analysis examples in the pipeline!
Real Life Examples
H → μμ

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
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)

30% Speedup achieved (21 KHz) thanks to small buffer optimisation (not in ROOT 6.16)

Realistic Analysis, Large Computation Graph: Good Performance & Efficient Scaling
192*2 cores Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz

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!
Investigate and prototype a complement to PROOF

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

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

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

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

JavierCVilla/PyRDF

Not in 6.16
Working prototype available!
Distributed Systems

Original RDataFrame

```python
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()
```

RDataFrame with PyRDF

```python
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()
```

Spark backend selected by default
RDF+Spark Scaling

- Revisited published TOTEM analysis
- CS thesis about this effort
- It works and there is room for further improvement
Array Syntax
Ergonomic interfaces for treating collections: a must for HEP data analysis

ROOT::RVec class:
- std::vector like interface
- Array operations are vectorised
- Math functions supported
- Can adopt memory
- Small buffer optimisation

RDataFrame relies on RVec for treating collections
- Zero copy with adoption
- SBO makes functional approach performant
ROOT::RVec<T> In Action

RVec<double> mus_pt {15., 12., 10.6, 2.3, 4., 3.};
RVec<double> mus_eta {1.2, -0.2, 4.2, -5.3, 0.4, -2.};
RVec<double> good_mus_pt = mus_pt[mus_pt > 10 && abs(mus_eta) < 2.1];

RVec<float> vals = {2.f, 5.5f, -2.f};
RVec<float> sin_vals = sin(vals);

ROOT::EnableImplicitMT();
RDataFrame f(treename, filename);
f.Define("good_pt", "sqrt(px*px + py*py)[E>100]")
 .Histo1D("pt", "pt", 16, -.5, 3.5, "good_pt")->Draw();
Keywords, Actions and Transformations
Transformations allow to modify the dataset.

<table>
<thead>
<tr>
<th>Transformation</th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Define</td>
<td>Creates a new column in the dataset.</td>
</tr>
<tr>
<td>DefineSlot</td>
<td>Same as Define, but the user-defined function must take an extra unsigned \texttt{int slot} as its first parameter. \texttt{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 Define transformation when using \texttt{RDataFrame} after \texttt{ROOT::EnableImplicitMT()}. DefineSlot works just as well with single-thread execution; in that case \texttt{slot} will always be 0.</td>
</tr>
<tr>
<td>DefineSlotEntry</td>
<td>Same as DefineSlot, but the entry number is passed in addition to the slot number. This is meant as a helper in case some dependency on the entry number needs to be honored.</td>
</tr>
<tr>
<td>Filter</td>
<td>Filter the rows of the dataset.</td>
</tr>
<tr>
<td>Range</td>
<td>Creates a node that filters entries based on range of entries</td>
</tr>
</tbody>
</table>
## Lazy actions do not trigger the event loop

<table>
<thead>
<tr>
<th>Lazy action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregate</td>
<td>Execute a user-defined accumulation operation on the processed column values.</td>
</tr>
<tr>
<td>Book</td>
<td>Book execution of a custom action using a user-defined helper object.</td>
</tr>
<tr>
<td>Cache</td>
<td>Caches in contiguous memory columns' entries. Custom columns can be cached as well, filtered entries are not cached. Users can specify which columns to save (default is all).</td>
</tr>
<tr>
<td>Count</td>
<td>Return the number of events processed.</td>
</tr>
<tr>
<td>Display</td>
<td>Obtains the events in the dataset for the requested columns. The method returns a RDisplay instance which can be queried to get a compressed tabular representation on the standard output or a complete representation as a string.</td>
</tr>
<tr>
<td>Fill</td>
<td>Fill a user-defined object with the values of the specified branches, as if by calling <code>Obj.Fill(branch1, branch2, ...)</code>.</td>
</tr>
<tr>
<td>Graph</td>
<td>Fills a TGraph with the two columns provided. If Multithread is enabled, the order of the points may not be the one expected, it is therefore suggested to sort if before drawing.</td>
</tr>
<tr>
<td>Histo{1D,2D,3D}</td>
<td>Fill a {one,two,three}-dimensional histogram with the processed branch values.</td>
</tr>
<tr>
<td><strong>Max</strong></td>
<td>Return the maximum of processed branch values. If the type of the column is inferred, the return type is double, the type of the column otherwise.</td>
</tr>
<tr>
<td><strong>Mean</strong></td>
<td>Return the mean of processed branch values.</td>
</tr>
<tr>
<td><strong>Min</strong></td>
<td>Return the minimum of processed branch values. If the type of the column is inferred, the return type is double, the type of the column otherwise.</td>
</tr>
<tr>
<td><strong>Profile{1D,2D}</strong></td>
<td>Fill a {one,two}-dimensional profile with the branch values that passed all filters.</td>
</tr>
<tr>
<td><strong>Reduce</strong></td>
<td>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 branch. Return the final result of the reduction operation. An optional parameter allows initialization of the result object to non-default values.</td>
</tr>
<tr>
<td><strong>Report</strong></td>
<td>Obtains 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 RCutFlowReport instance which can be queried programmatically to get information about the effects of the individual cuts.</td>
</tr>
<tr>
<td><strong>StdDev</strong></td>
<td>Return the unbiased standard deviation of the processed branch values.</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td>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.</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Take</strong></td>
<td>Extract a column from the dataset as a collection of values. If the type of the column is a C-style array, the type stored in the return container is a <code>ROOT::VecOps::RVec&lt;T&gt;</code> to guarantee the lifetime of the data involved.</td>
</tr>
</tbody>
</table>
Instant actions do trigger the event loop

<table>
<thead>
<tr>
<th>Instant action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreach</td>
<td>Execute a user-defined function on each entry. Users are responsible for the thread-safety of this lambda when executing with implicit multi-threading enabled.</td>
</tr>
<tr>
<td>ForeachSlot</td>
<td>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.</td>
</tr>
<tr>
<td>Snapshot</td>
<td>Writes processed data-set 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. <strong>Snapshot can be made lazy</strong> setting the appropriate flag in the snapshot options.</td>
</tr>
</tbody>
</table>
Python Interoperability
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")
# Run input pipeline with C++ performance that can process TBs of data, reads from remote, ...

```python
text
import ROOT
df = ROOT.RDataFrame("tree", "file.root")
    .Filter("HLT_Mu22_v42", "Trigger requirement")
    .Filter("All(tight_iso)", "Quality cut")
    .Define("r", "sqrt(eta*eta + phi*phi)"")

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

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

---

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

See [A More Pythoic, Interoperable and Modern PyROOT](https://example.com), 11/3 16:10 Steinmatte
# 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("HLT_Mu22_v42", "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)
```

<table>
<thead>
<tr>
<th>r</th>
<th>eta</th>
<th>phi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.26</td>
<td>-0.5</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>-1.0</td>
</tr>
<tr>
<td>2</td>
<td>4.45</td>
<td>2.1</td>
</tr>
</tbody>
</table>

...
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, focussing 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)")

zero copy Py <-> Cpp through arrays
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