Declarative Analysis in ROOT with RDataFrame

S. Wunsch for the ROOT team
HSF Workshop Jefferson Lab

This Talk

- 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

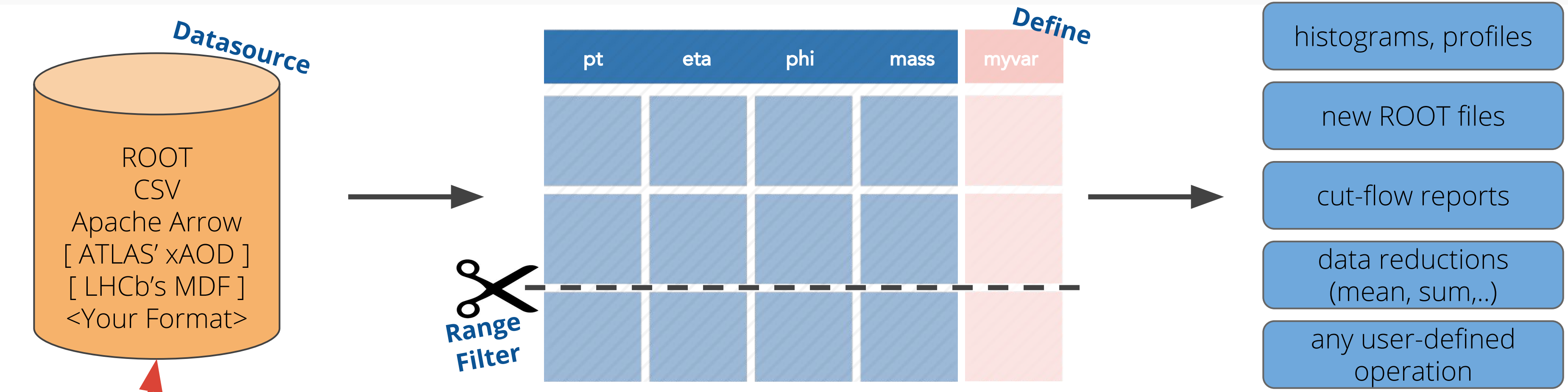
A recipe for efficient HEP analyses

➔ strive for a **simple programming model**

➔ expose modern, elegant interfaces that are **easy to use correctly** and hard to use incorrectly

➔ allow to **transparently benefit from parallelism**

ROOT Declarative Analysis: RDataFrame



Customisation point, public interface!

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]()

An ergonomic, fast C++ dataframe

`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 $r2 = x^2 + y^2$
`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

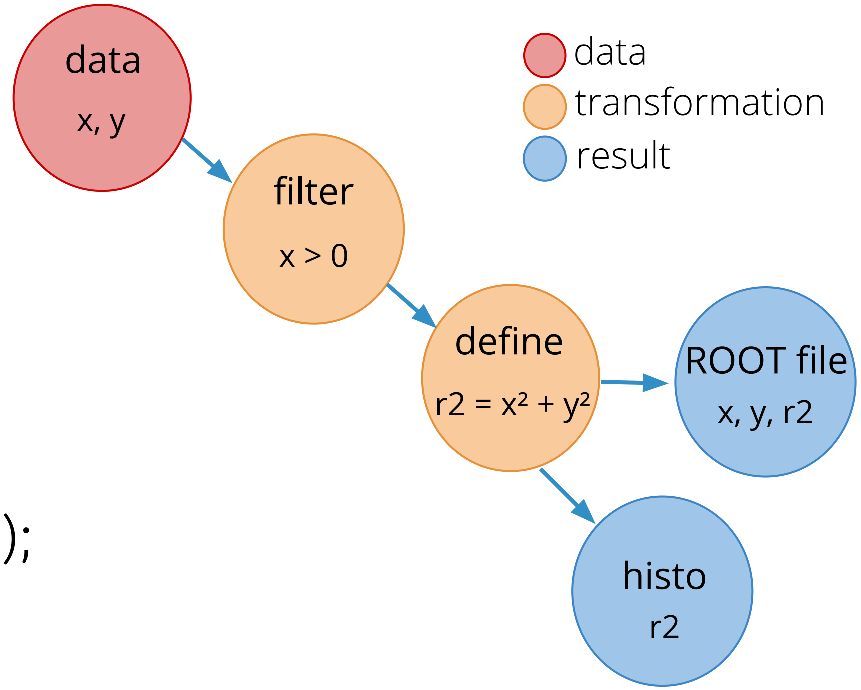
An ergonomic, fast C++ dataframe

```
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++ $\longrightarrow$ JIT

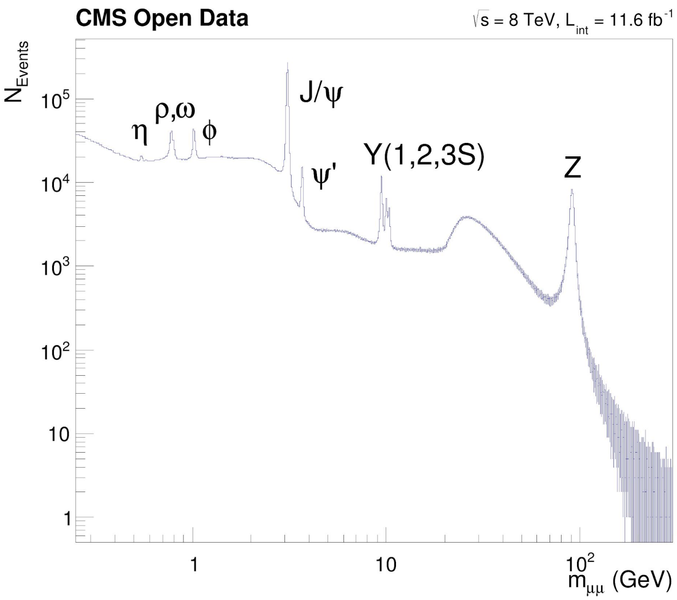
C++

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

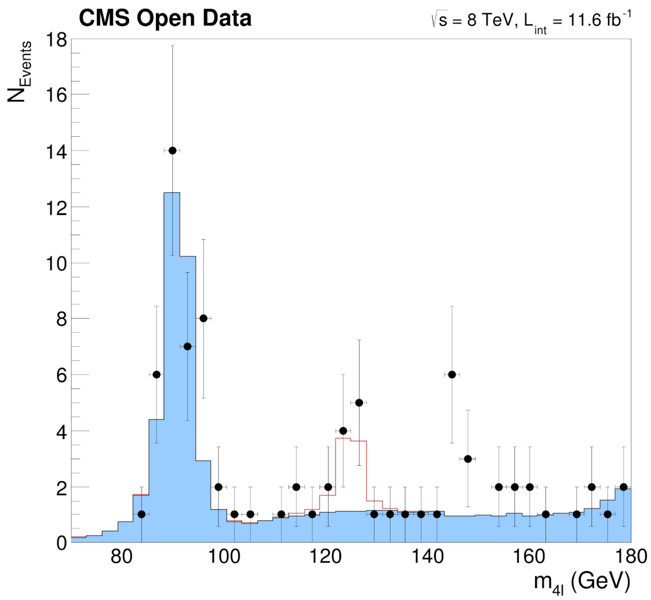
C++ with cling's just-in-time compilation

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

Examples from the tutorials



[Tutorial]



[Tutorial]

- Fully runnable examples with data and code
- More realistic analysis examples in the pipeline!

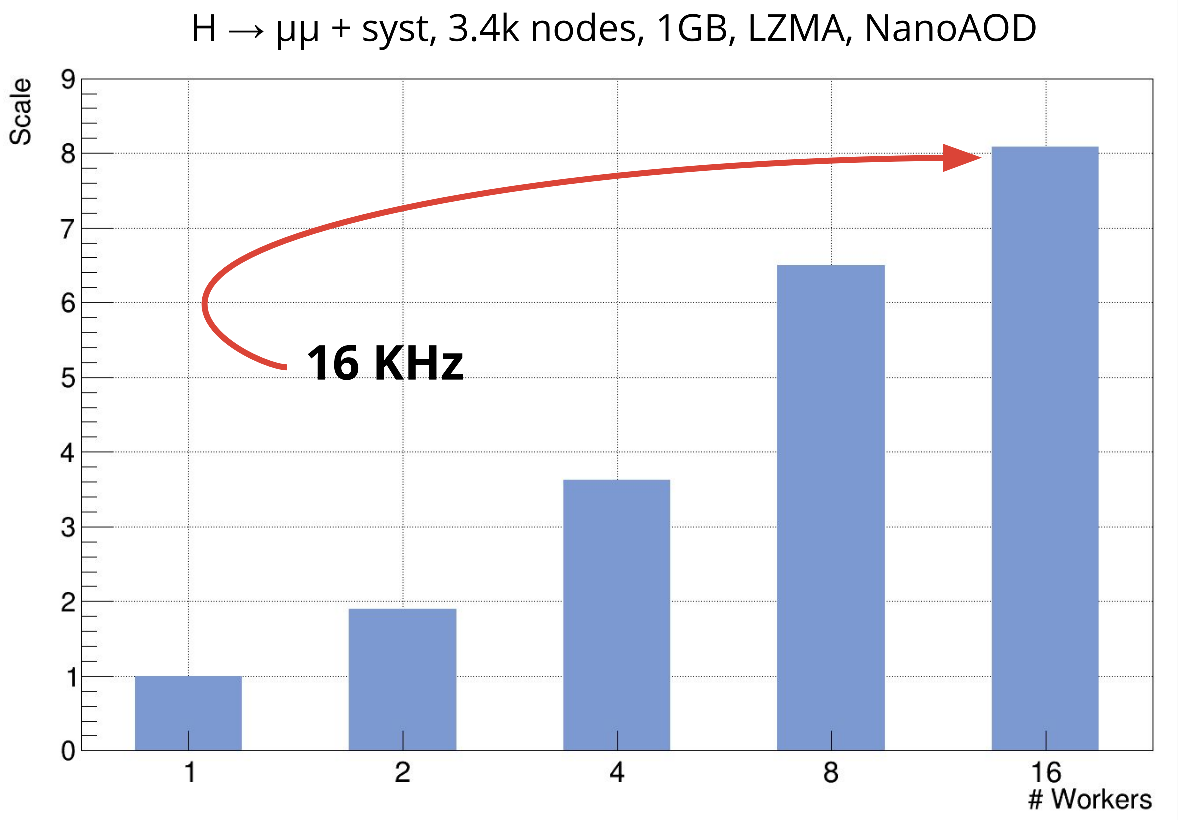
Real Life Examples

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)



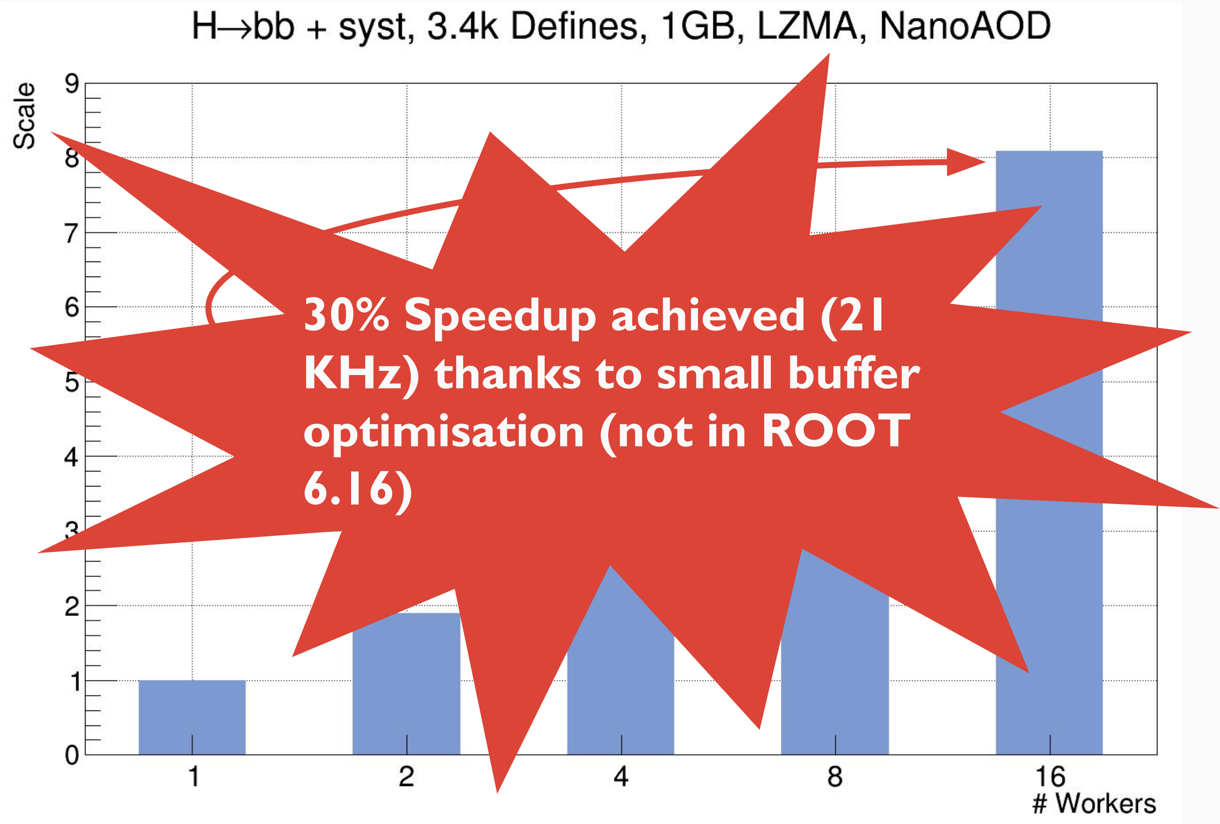
Realistic Analysis, Large Computation Graph: Good Performance & Efficient Scaling

H → bb

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

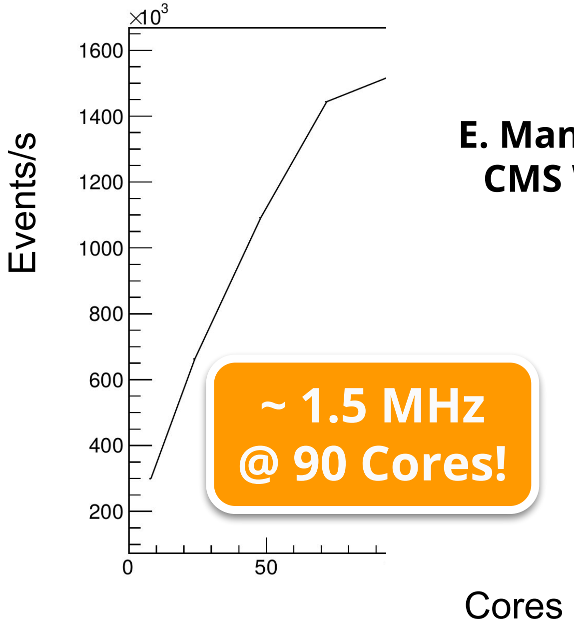
Intel Core i7 7820X (8*2 cores, 3.60GHz)



Realistic Analysis, Large Computation Graph: Good Performance & Efficient Scaling

Does All This Scale?

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



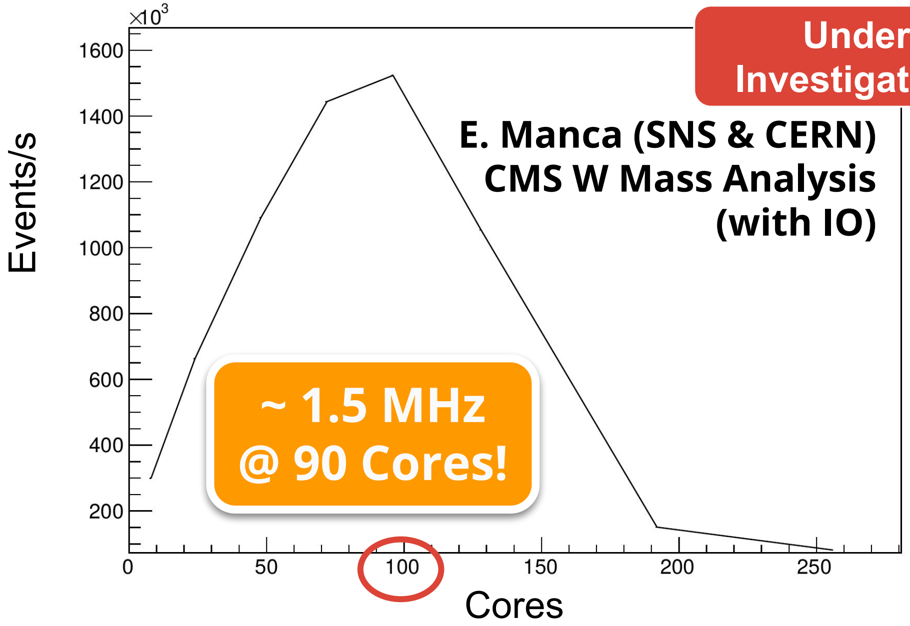
E. Manca (SNS & CERN)
CMS W Mass Analysis
(with IO)

~ 1.5 MHz @ 90 Cores!

RDataFrame Scales on Many Cores

Does All This Scale?

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



Under Investigation!

RDataFrame Scales on Many Cores

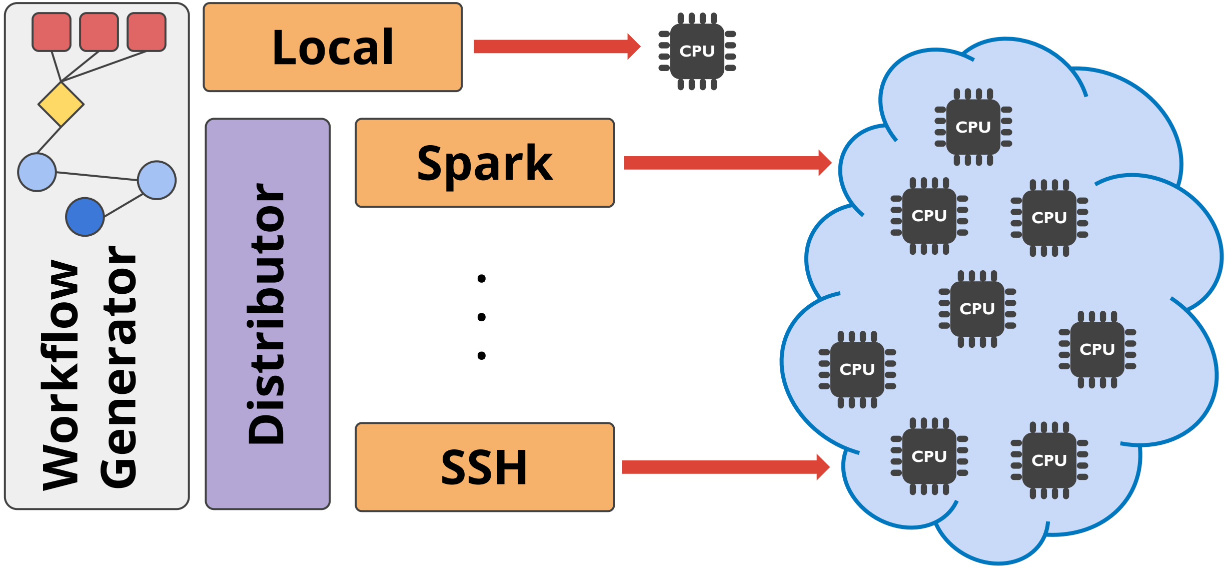
Distributed Analysis

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

```
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!

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()
```

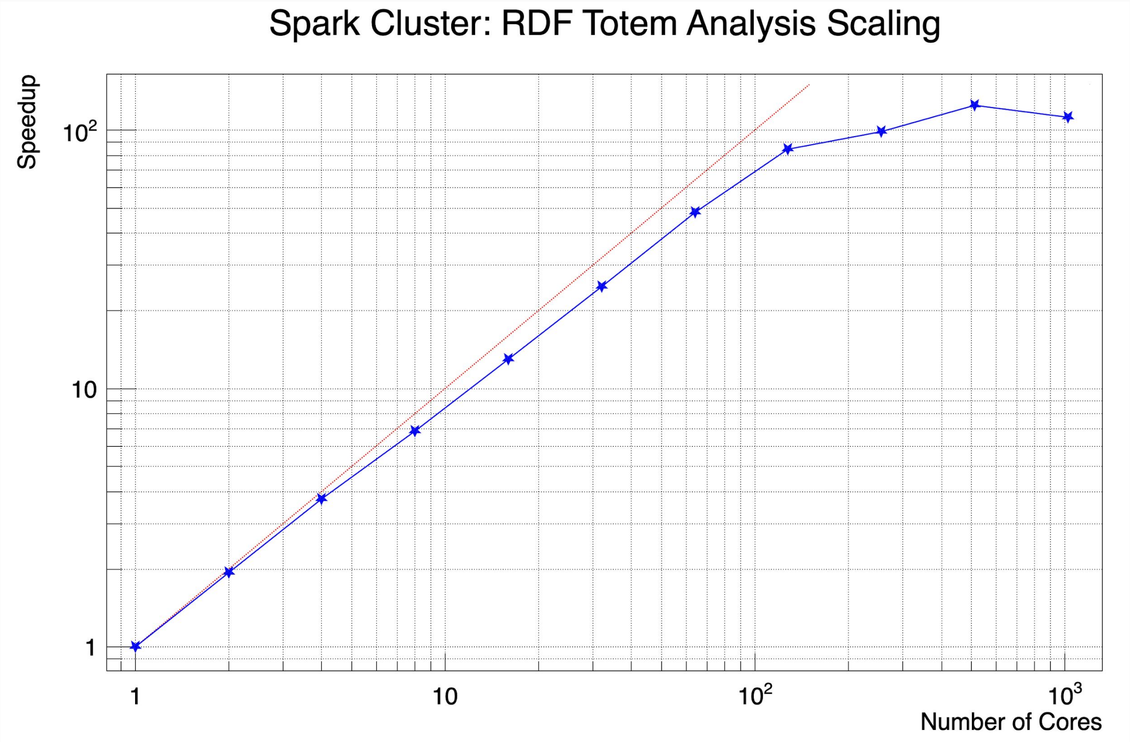
RDataFrame with PyRDF

Spark backend selected by default

```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()
```

RDF+Spark Scaling

- Revisited published TOTEM analysis
- CS thesis about this effort
- It works and there is **room for further improvement**



Array Syntax

Array Syntax and its Support

- 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);
```

Already integrated with RDataFrame

py is a collection, not a scalar

```
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

- Transformations allow to modify the dataset

Transformation	**Description*
Define	Creates a new column in the dataset.
DefineSlot	Same as `Define`, 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 `Define` transformation when using **`RDataFrame`** after **`ROOT::EnableImplicitMT()`**. `DefineSlot` works just as well with single-thread execution: in that case `slot` will always be `0`.
DefineSlotEntry	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 honoured.
Filter	Filter the rows of the dataset.
Range	Creates a node that filters entries based on range of entries

Lazy Actions 1/3

- Lazy actions do not trigger the event loop

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	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).
Count	Return the number of events processed.
Display	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.
Fill	Fill a user-defined object with the values of the specified branches, as if by calling \`Obj.Fill(branch1, branch2, ...).
Graph	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.
Histo{1D,2D,3D}	Fill a {one,two,three}-dimensional histogram with the processed branch values.

Lazy Actions 2/3

Max	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.
Mean	Return the mean of processed branch values.
Min	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.
Profile{1D,2D}	Fill a {one,two}-dimensional profile with the branch 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 branch. Return the final result of the reduction operation. An optional parameter allows initialization of the result object to non-default values.
Report	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.
StdDev	Return the unbiased standard deviation of the processed branch values.

Lazy Actions 3/3

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. If the type of the column is a C-style array, the type stored in the return container is a `ROOT::VecOps::RVec`<T> to guarantee the lifetime of the data involved.

Instant Actions 3/3

- Instant actions do trigger the event loop

Instant action	Description
Foreach	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.
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	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. `Snapshot` can be made *lazy* setting the appropriate flage in the snapshot options.

Python Interoperability

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")
```

RDataFrame to numpy and pandas

```python
# Run input pipeline with C++ performance that can process TBs of data, reads from remote, ...
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)

   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

RDataFrame to numpy and pandas

```
# Run input pipeli                                            te, ...
import ROOT
df = ROOT.RDataFra
          .Filter("
          .Filter("
          .Define("

# Extract selectio
col_dict = df.AsNu

# Wrap data with p
import pandas
p = pandas.DataFra
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
...
```

RDF +
ert to
6.16/02

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

Python Callables in RDF

```python
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)")
```

R&D

Make Python callable available to the Cling

- Compilation with Numba also possible
- Functionality available, focussing on the interfaces and programming model

NumPy Arrays as RDataSource

```python
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)")
```

R&D

A factory function returning a RDataFrame

zero copy Py <-> Cpp through arrays

Wrap-up

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