

PyRDF: Distributed RDataFrame

Summary of the current status

V. E. Padulano, J. Cervantes, E. Tejedor





Outline

- ▶ What is it?
- ▶ New Features
- ▶ Tests
- ▶ Use cases



What is it?



Started as a GSOC project in 2018

[Python layer](#) on top of RDataFrame



Enable RDataFrame analysis to run with [distributed resources](#)

PyROOT to access ROOT from Python

Spark backend

PyRDF

PyROOT

ROOT / RDF

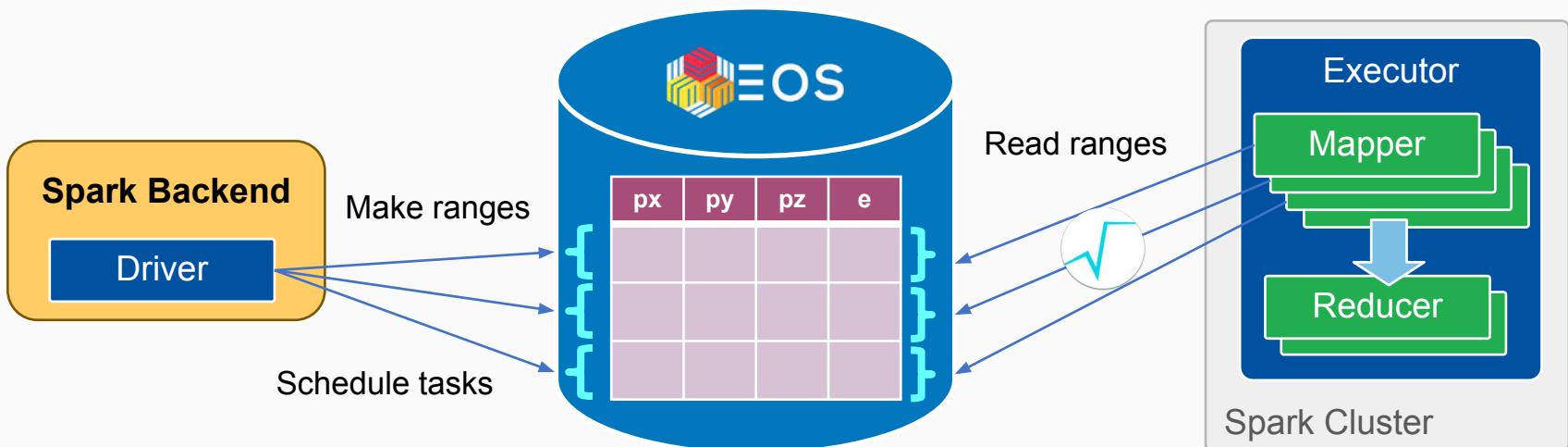


[PyRDF repo](#)



Spark backend

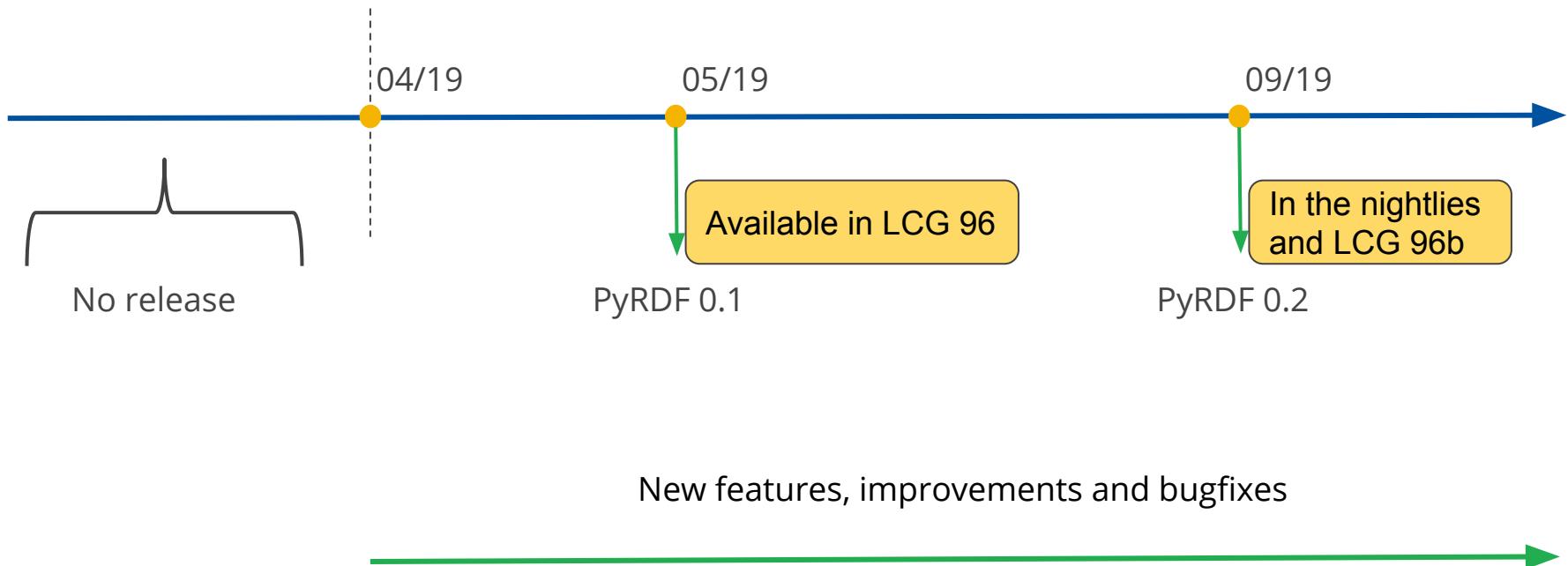
- ▶ **Map-reduce** workflow where every mapper runs the RDataFrame computation graph on a range of collision events
- ▶ Spark provides APIs in Scala, Java, **Python**
- ▶ Run **analysis in C++ with Spark**
 - Exploiting its python API and PyROOT



New Features



New releases





Feature Evolution

At my arrival:

- ▶ Minimal implementation
- ▶ Only python 2
- ▶ Unstable graph pruning
- ▶ No official docs
- ▶ Missing RDF operations

PyRDF 0.1:

- ▶ Functional implementation
- ▶ First PyRDF release on LCG
- ▶ Full Python 3 support
- ▶ Stable and improved graph pruning
- ▶ Send C++ headers and shared libraries to Spark

PyRDF 0.2:

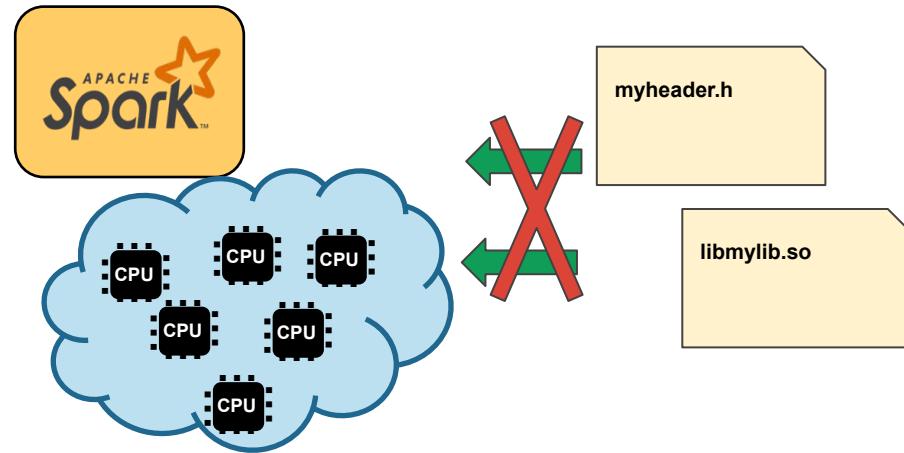
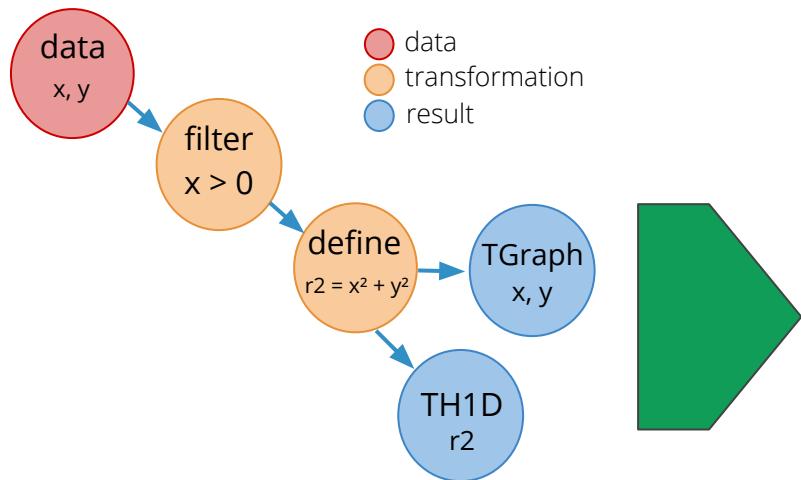
- ▶ Add friend trees compatibility
- ▶ Add logging capabilities
- ▶ Official docs created
- ▶ Many RDF operations implemented with Spark



C++ headers to distributed executors

- ▶ ROOT allows user-defined headers and shared libraries
- ▶ But they are not sent to the Spark executors at runtime

RDataFrame computational graph





C++ headers to distributed executors

myheader.h

```
#ifndef myheader  
#define myheader  
  
bool f(int num){  
    return num < 5;  
}  
  
#endif
```

Declare locally

Sent to executors
at runtime

```
import PyRDF  
PyRDF.use("spark")  
PyRDF.include_headers("myheader.h")  
  
df = PyRDF.RDataFrame(256)  
  
df.Filter("f(rdfentry_)).Count()
```

```
[...]  
header = SparkFiles.get("myheader.h")  
Utilsdeclare_headers(header)  
[...]
```

Also working with directories of headers
and shared libraries

Users can send headers at runtime
before: only at connection time



More operations available

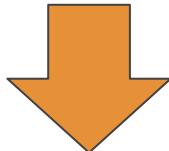
Many operations which were previously unavailable in a distributed environment now are:

- ▶ Count, Sum, statistics in general (through Stats)
- ▶ Snapshot
- ▶ AsNumpy



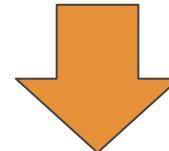
Snapshot

```
import ROOT  
df = ROOT.RDataFrame(10000)\\  
    .Define("x","rdfentry_")\\  
    .Define("y","x*x")  
  
df.Snapshot("treename","file.root")
```



Save to a file in the [local](#) machine.

```
import PyRDF  
PyRDF.use("spark")  
df = PyRDF.RDataFrame(10000)\\  
    .Define("x","rdfentry_")\\  
    .Define("y","x*x")  
  
df.Snapshot("treename","file.root")
```



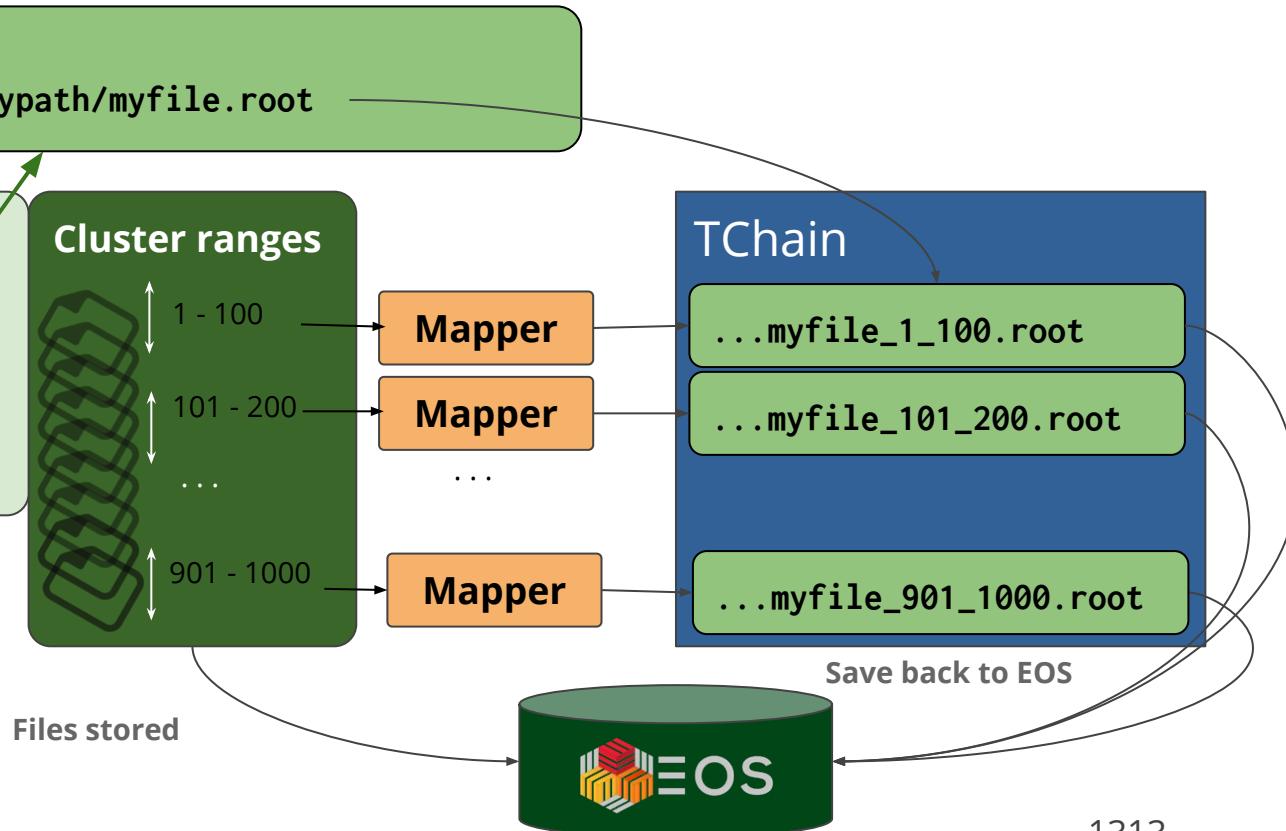
Save the distributed chunks of data to an [EOS](#) path the user decides

Snapshot

Path to a EOS file:

root://eosuser.cern.ch//mypath/myfile.root

```
import PyRDF  
PyRDF.use("spark")  
[RDF operations...]  
df.Snapshot(eospath)
```



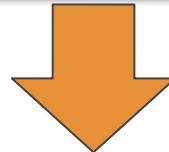


AsNumpy

```
import ROOT
df = ROOT.RDataFrame(10000) \
    .Define("x","rdfentry_") \
    .Define("y","x*x")
npy = df.AsNumpy()
```



```
import PyRDF
PyRDF.use("spark")
df = PyRDF.RDataFrame(10000) \
    .Define("x","rdfentry_") \
    .Define("y","x*x")
npy = df.AsNumpy()
```



A **collection** (dictionary) of **numpy arrays** corresponding to the columns of the RDataFrame



- ▶ Documentation is built with sphinx
- ▶ Automatically created from docstrings
- ▶ Hosted on [Read the Docs!](#)

The screenshot shows the PyRDF documentation page on the Read the Docs platform. The top navigation bar includes the PyRDF logo, a search bar labeled "Search docs", and links for "Docs" and "View page source". The main content area has a dark header with "CONTENTS:" and a list of sections: "The PyRDF API reference", "PyRDF's supported backends", and "PyRDF's utility functions". Below this, the main title "PyRDF : The Python ROOT DataFrame Library" is displayed in bold. A brief description follows: "A pythonic wrapper around ROOT's `RDataFrame` with support for distributed execution." A "Sample usage:" section contains a code snippet:

```
import PyRDF, ROOT
PyRDF.use('spark', {'npartitions':4})

df = PyRDF.RDataFrame("data", ['https://root.cern/files/teaching/CMS_Open_Dataset.root'])

etaCutStr = "fabs(eta1) < 2.3"
df_f = df.Filter(etaCutStr)

df_histogram = df_f.Histo1D("eta1")

canvas = ROOT.TCanvas()
df_histogram.Draw()
canvas.Draw()
```



Logging PyRDF

- Logs implemented with Python standard module `logging`
- Create logger in the Python script

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
[add handlers and customize formatting]
```

```
import PyRDF
logger = PyRDF.create_logger("DEBUG")
[ RDF stuff ... ]
```

```
DEBUG: PyRDF.RDataFrame[2019-09-15 17:11:44,779]: Created RDataFrame head node and proxy
DEBUG: PyRDF.Proxy[2019-09-15 17:11:44,780]: Created new Define node
DEBUG: PyRDF.Proxy[2019-09-15 17:11:44,780]: Created new Count node[
DEBUG: PyRDF.Node[2019-09-15 17:11:44,787]: Starting computational graph pruning
DEBUG: PyRDF.Node[2019-09-15 17:11:44,788]: Checking Define node for pruning
DEBUG: PyRDF.Node[2019-09-15 17:11:44,788]: Checking Count node for pruning
DEBUG: PyRDF.Node[2019-09-15 17:11:44,788]: Count node shouldn't be pruned
DEBUG: PyRDF.Node[2019-09-15 17:11:44,788]: Define node shouldn't be pruned
DEBUG: PyRDF.Node[2019-09-15 17:11:44,788]: Graph pruning completed
DEBUG: PyRDF.backend.Local[2019-09-15 17:11:45,237]: Action operations in the loop:
['Count']
```



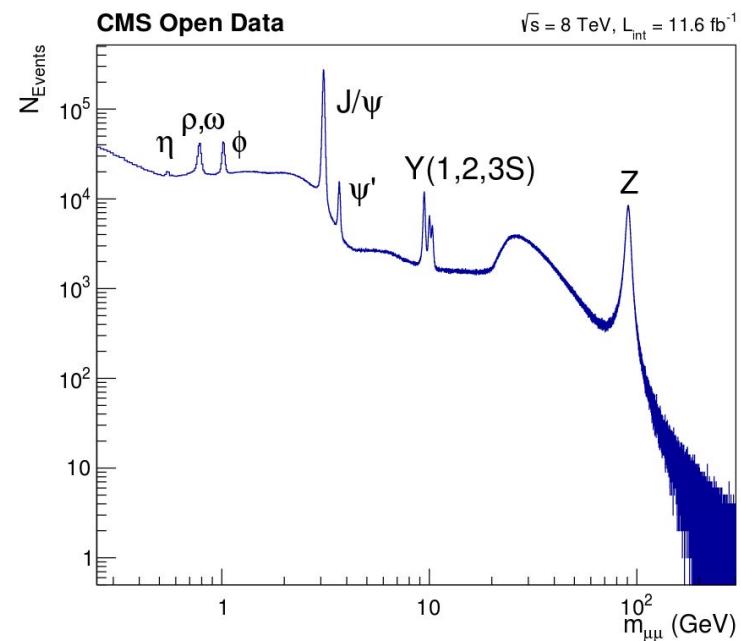
Tests



Dimuon Analysis Tests

A comparison between different configurations for running the dimuon [tutorial](#) on SWAN:

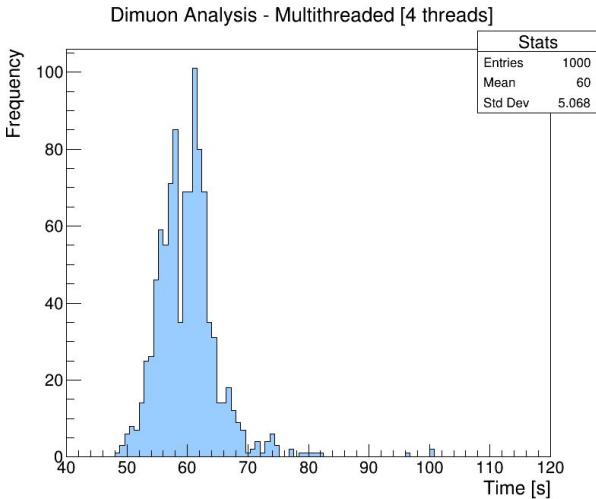
- ▶ locally, with 4 threads enabled
- ▶ on the spark executors (1 core each)
 - 4 executors
 - 8 executors
 - 16 executors
 - 32 executors



Dimuon Analysis Tests

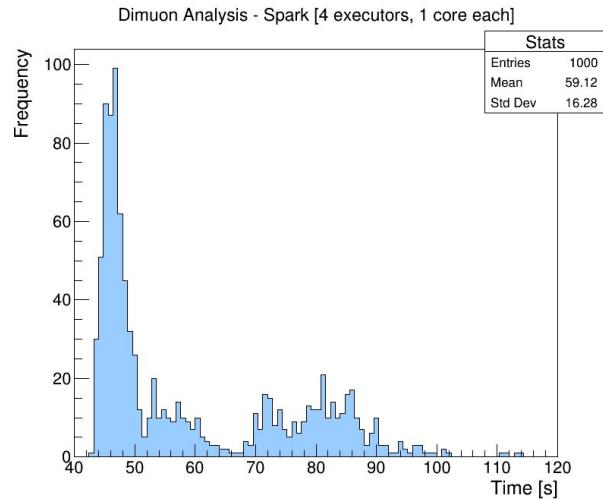
Multithreaded [4 threads]

- ▶ AVG time [s]: 60.5 +- 12.7
- ▶ Most frequent time [s]: 61.2



Spark [4 executors, 1 core each]

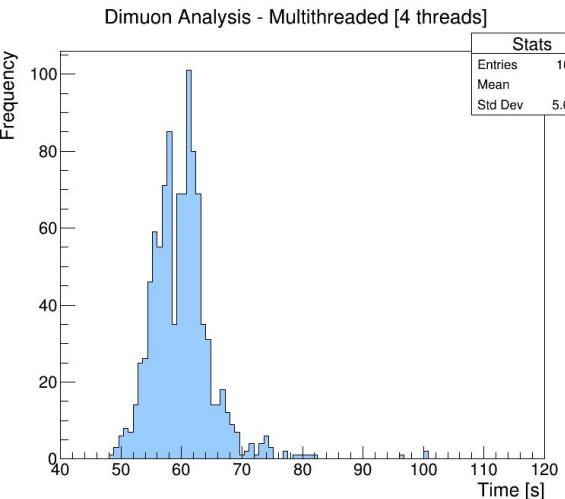
- ▶ AVG time [s]: 59.1 +- 32.5
- ▶ Most frequent time [s]: 46.8



Dimuon Analysis Tests

Multithreaded [4 threads]

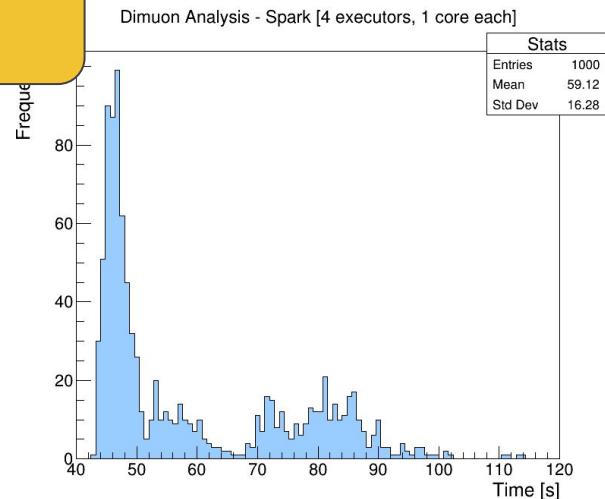
- ▶ AVG time [s]: 60.5 +- 12.7
- ▶ Most frequent time [s]: 61.2



Spark [4 executors, 1 core each]

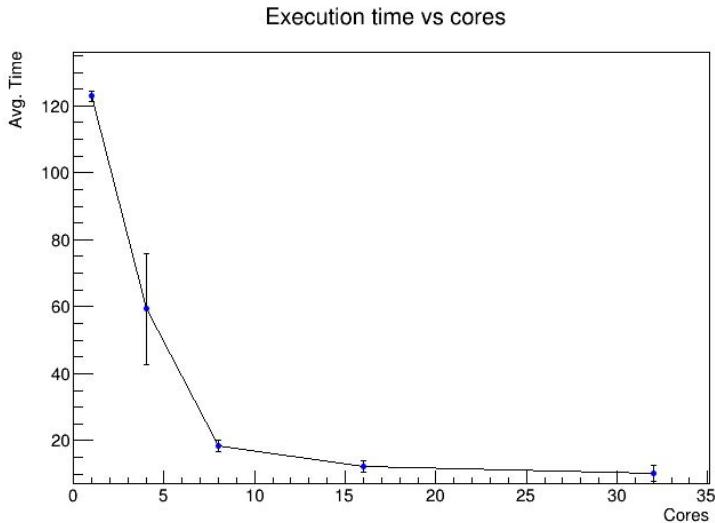
- ▶ AVG time [s]: 59.1 +- 32.5
- ▶ Most frequent time [s]: 46.8

Most frequent time is much lower with Spark





Dimuon Analysis Tests



- ▶ Good scaling up to 8 cores
- ▶ But different scaling factor
- ▶ Speedup ~ 1 after 8 cores

Use cases





ATLAS use case

ATLASSM_WBR > FriendTTree_test
Last Checkpoint: 05/22/2019 (unsaved changes)

FILE EDIT VIEW INSERT CELL KERNEL WIDGETS HELP

Code



Simple PyRDF test for the Friend TTree functionality

Install latest PyRDF version

```
In [1]: # # Configuration to the SparkContext is added via the SWAN interface as follows:  
# # https://github.com/JavierCVilla/PyRDF/tree/master/demos  
import sys  
sys.path.insert(0, "/eos/user/v/vpadulan/SWAN_projects/newPy/PyRDF")  
sys.path.append("/eos/user/v/vpadulan/SWAN_projects/ATLASSM_WBR/HAPPY")  
sys.path.append("/eos/user/v/vpadulan/SWAN_projects/ATLASSM_WBR/xTauReader")  
# # Add python module (temporal)  
sc.addPyFile("../PyRDF.zip")
```

Configure

```
In [2]: import ROOT  
import PyRDF  
# PyRDF = ROOT  
  
%jsroot on  
  
PyRDF.use("spark")
```

Welcome to JupyterROOT 6.19/01

Base TTree: define and plot

```
In [3]: baseTreeOrg = ROOT.TChain( 'NOMINAL' )  
  
# File from the project  
baseTreeOrg.Add( "root://eosuser.cern.ch//eos/user/d/dponomar/SWAN_projects/ATLASSM_WBR/Data/  
  
baseTree = baseTreeOrg.CloneTree(2019, "fast")  
baseTree.SetEstimate( baseTree.GetEntries() + 1 )
```



ATLAS user with a ROOT DataFrame based analysis



Intricated package with multiple dependencies

New Feature developed:
Friend trees with a Spark backend!



NanoAOD analysis to numpy arrays

CMS user developing his own physics analysis within a python framework.

- ▶ Analysis translated from traditional ROOT to `RDataFrame`
 - ▶ PyRDF used to connect to the `Spark` cluster and retrieve `numpy` arrays after `column definition` and `filtering`

Very early stage, but this approach can be easily extended to any NanoAOD analysis.



preliminary presentation, analysis repo



A challenging use case

Data from the TOTEM experiment

500 TB
RECO DATA
AOD DATA FORMAT



- ▶ Analysis requires [CMSSW](#) framework to run.
- ▶ SWAN and the Spark clusters take software from CVMFS, but only include the [SFT](#) repository.

Currently working towards [integrating](#) CMSSW in SWAN (first successful attempt in the user session) and the Spark executors.



[repo](#)



Thank you!



FriendTree Compatibility

During map phase:

1. Retrieve info about friend trees
2. Friend trees share same data range as their main tree
3. Build a chain of friend trees and add it to the main tree.

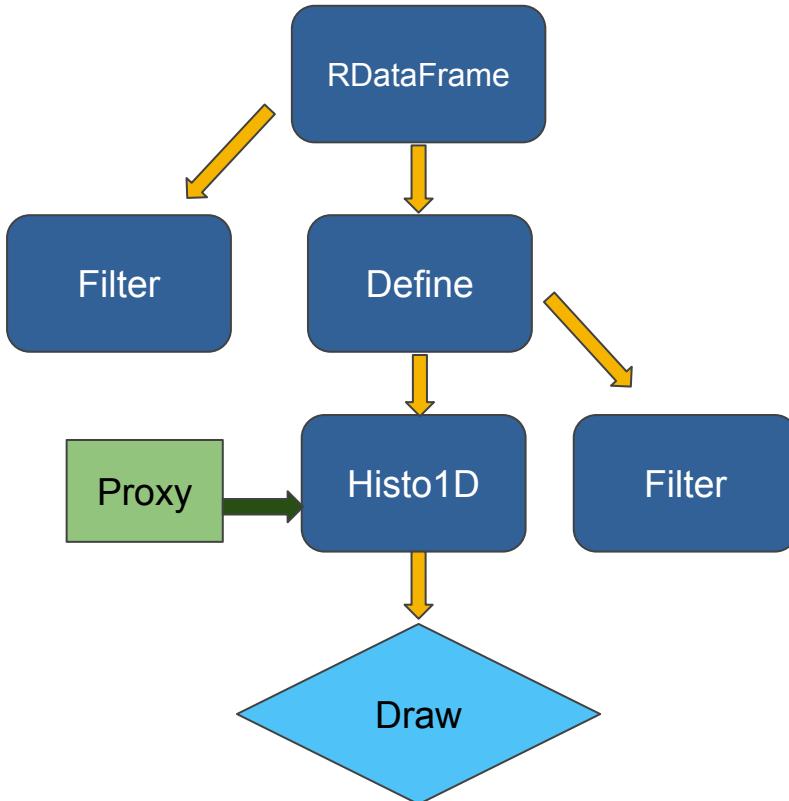
```
def get_friend_info(TTree):
    friends = TTree.GetListOfFriends()
    for friend in friends:
        friend_tree = friend.GetTree() # ROOT.TTree
        real_name = friend_tree.GetName()
        friend_filename = friend_tree.GetCurrentFile()
            .GetName()
```

```
[...]
```

```
# In map function
if friend_info:
    for friend_treename, friend_filenames in tree_files_names:
        # Start a TChain with the current friend treename
        friend_chain = ROOT.TChain(friend_treename)
        # Add each corresponding file to the TChain
        for filename in friend_filenames:
            friend_chain.Add(filename)
        # Set cache on the same range as the parent TChain
        friend_chain.SetCacheEntryRange(start, end)
        # Finally add friend TChain to the parent
        chain.AddFriend(friend_chain)
```



Graph Pruning



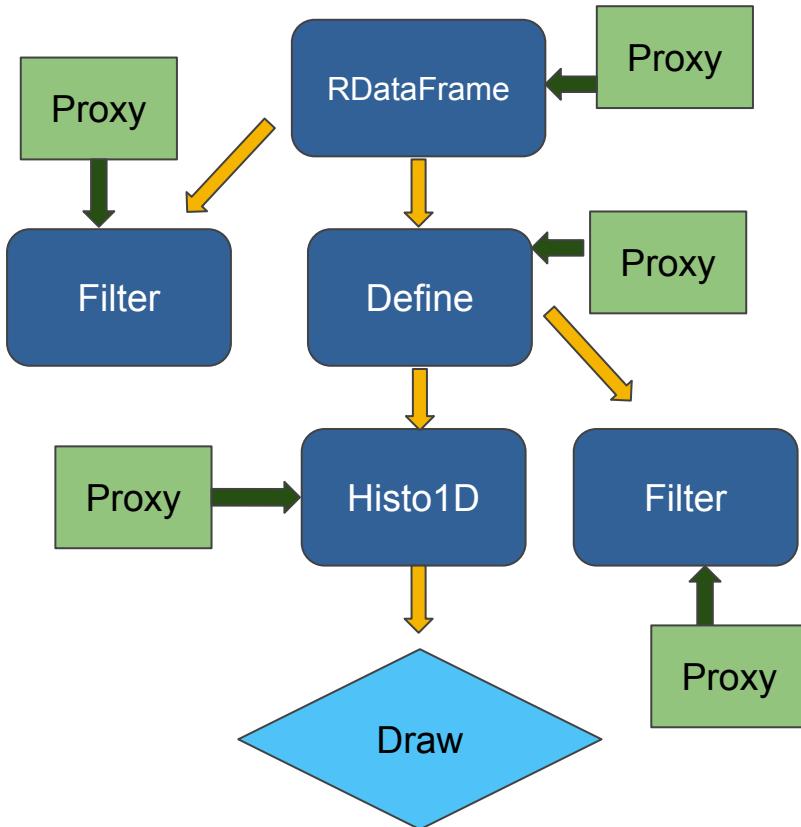
Before release 0.1 :

1. Check if node has children
2. If it has, check the children first
3. Check how many python objects are referencing the node
4. Check if the node is an action with an already computed value
5. If either (3) or (4) are true then the node can be pruned
6. Repeat on all the children

The condition on the number of referrers was very unstable!



Graph Pruning



After release 0.1 :

1. Check if node has children
2. If it has, check the children first
3. Check if the proxy flagged the node to be non referenced by the user
4. Check if the node is an action with an already computed value
5. If either (3) or (4) are true then the node can be pruned
6. Repeat on all the children

Now we can know precisely when the user doesn't need a node operation anymore