PyRDF: Distributed RDataFrame

Summary of the current status

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

ROOT

Data Analysis Framework

https://root.cern

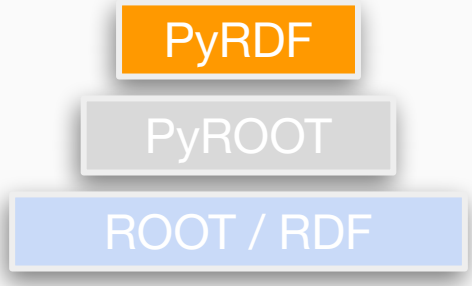
Outline

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

What is it?

PyRDF

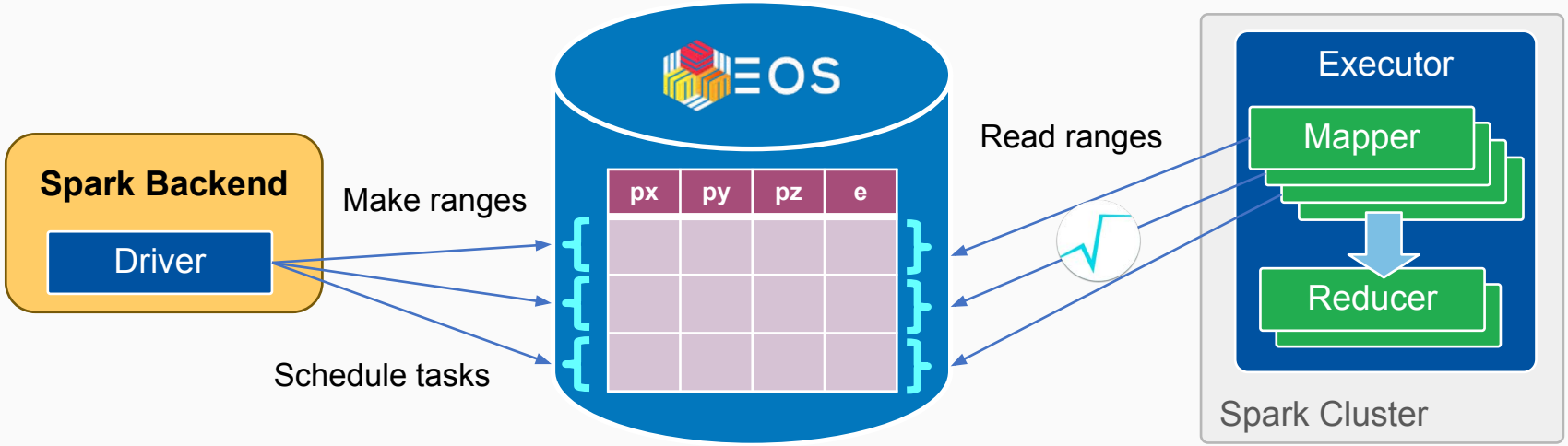
- Started as a GSOC project in 2018
- Python layer on top of RDataFrame
- Enable RDataFrame analysis to run with distributed resources
- PyROOT to access ROOT from Python
- Spark backend



PyRDF repo

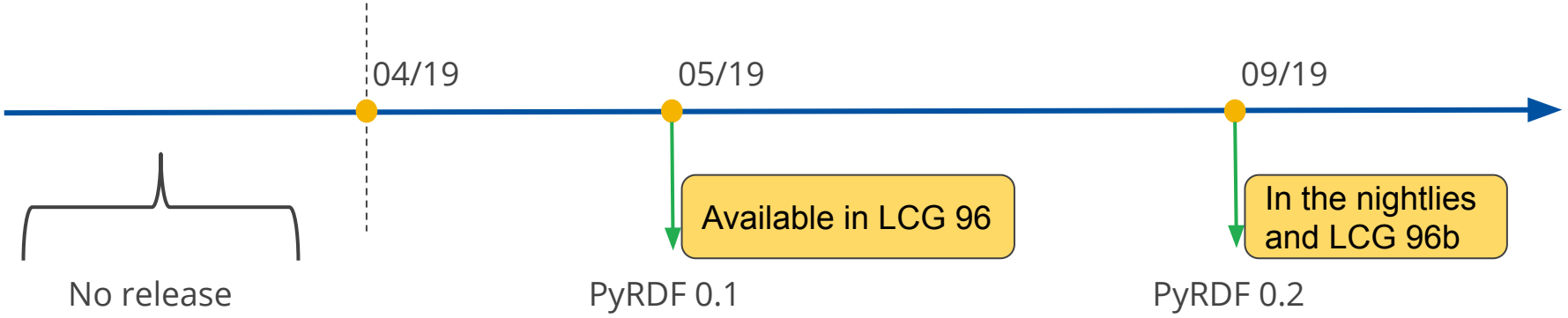
Spark backend

- **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



New features, improvements and bugfixes

Feature Evolution

At my arrival:

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

PyRDF 0.1:

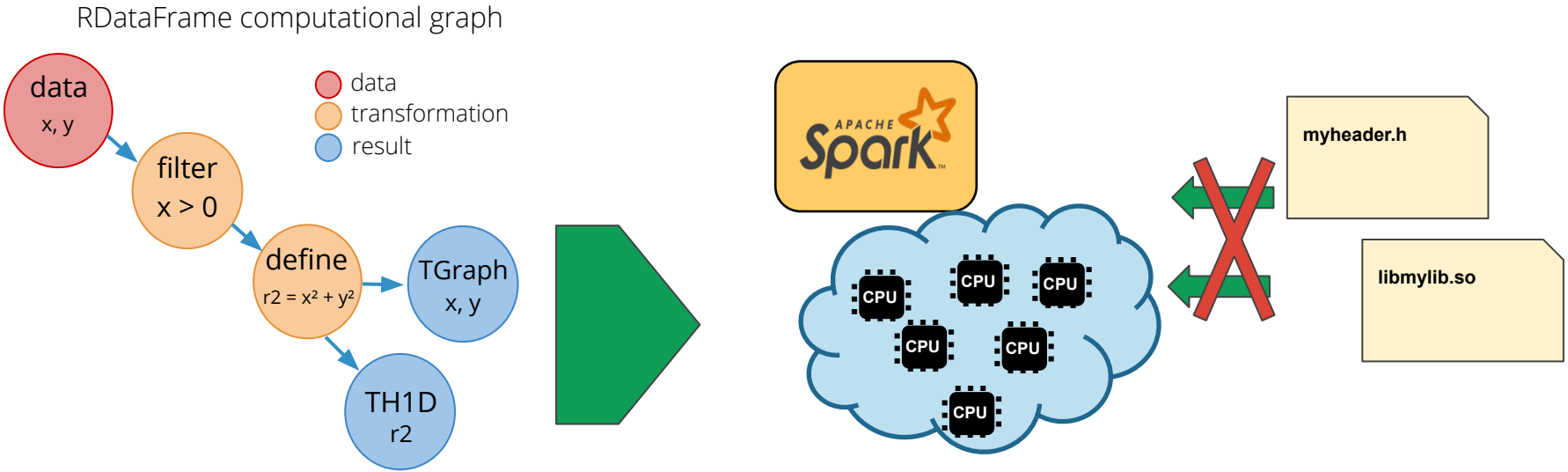
- Functional implementation
- First PyRDF release on LCG
- Full Python 3 support
- Stable and improved graph pruning
- Send C++ headers and shared libraries to Spark

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



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

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

```
import PyRDF
PyRDF.use("spark")
PyRDF.include_headers("myheader.h")

df = PyRDF.RDataFrame(256)

df.Filter("f(rdfentry_)").Count()
```

Also working with directories of headers and shared libraries

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

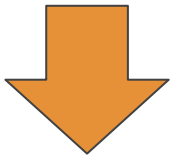
More operations available

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

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

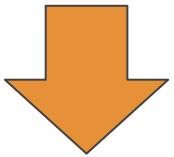
Snapshot

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



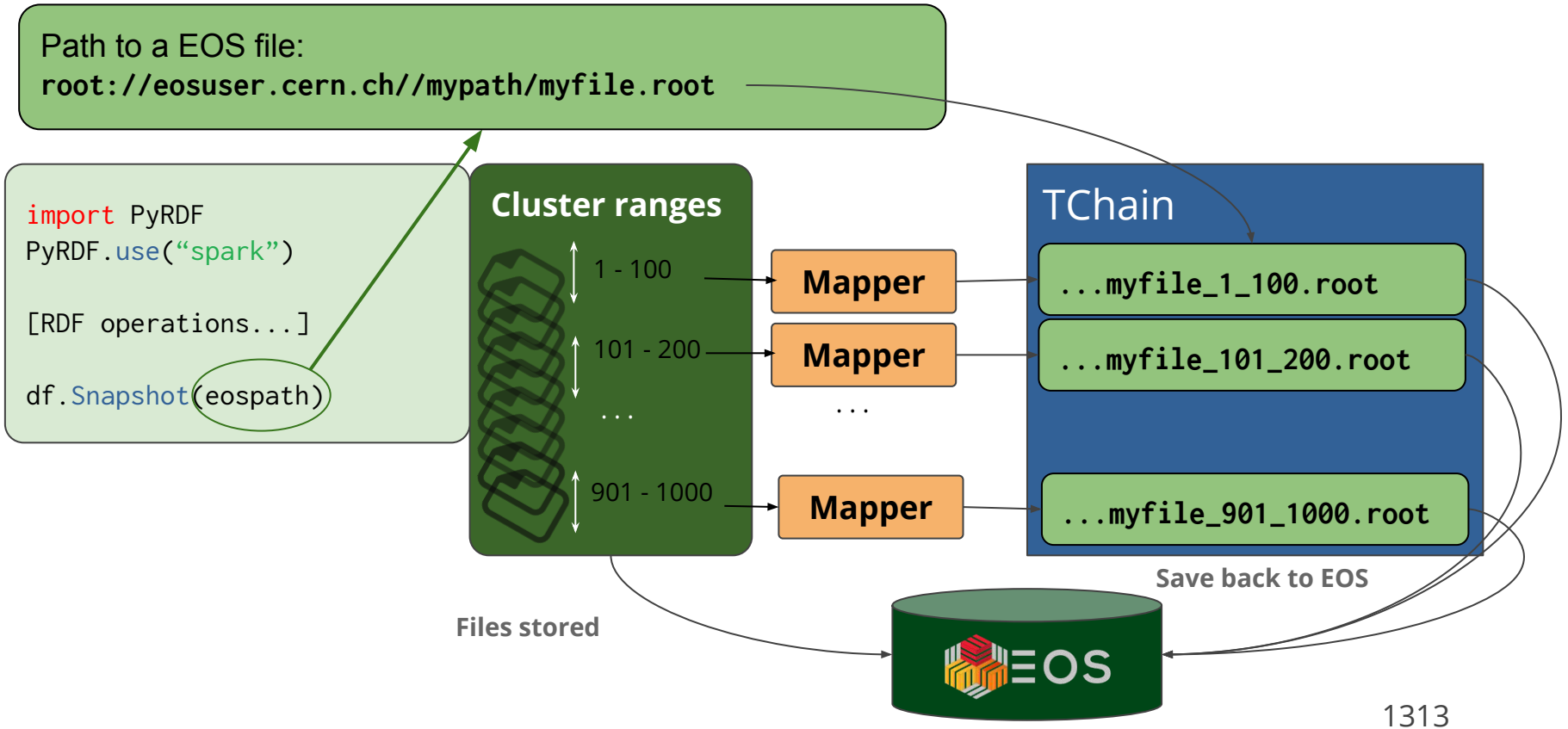
Save to a file in the local machine.

```
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)
Cluster ranges
1 - 100
101 - 200
...
901 - 1000
Mapper
Mapper
...
Mapper
TChain
...myfile_1_100.root
...myfile_101_200.root
...myfile_901_1000.root
Save back to EOS
Files stored
EOS

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

Docs

- Documentation is built with sphinx
- Automatically created from docstrings
- Hosted on Read the Docs!

Logging PyRDF

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

```python
import logging
logger = logging.getLogger()

logger.setLevel(logging.DEBUG)

[add handlers and customize formatting]
```

```python
import PyRDF
logger = PyRDF.create_logger("DEBUG")

[ RDF stuff … ]
```

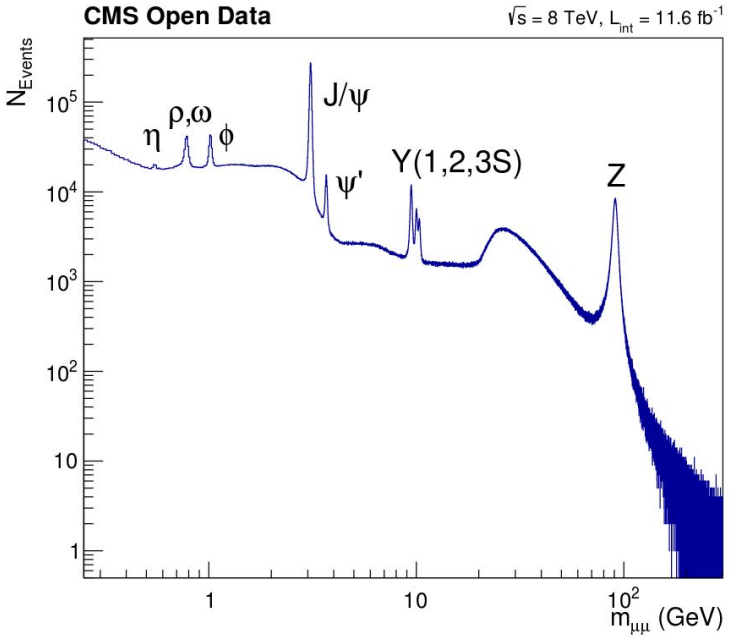
```
DEBUG: PyRDF.RDataFrame[2019-09-15 17:11:44,779]: Created RDataFrame head node and proxy
DEBUG: PyRDF.Proxy[2019-09-15 17:11:44,780]: Created new Define node
DEBUG: PyRDF.Proxy[2019-09-15 17:11:44,780]: Created new Count node
DEBUG: PyRDF.Node[2019-09-15 17:11:44,787]: Starting computational graph 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



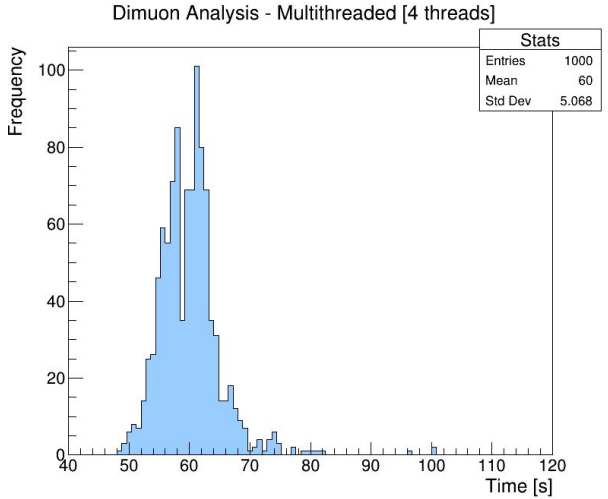
Dimuon Analysis Tests

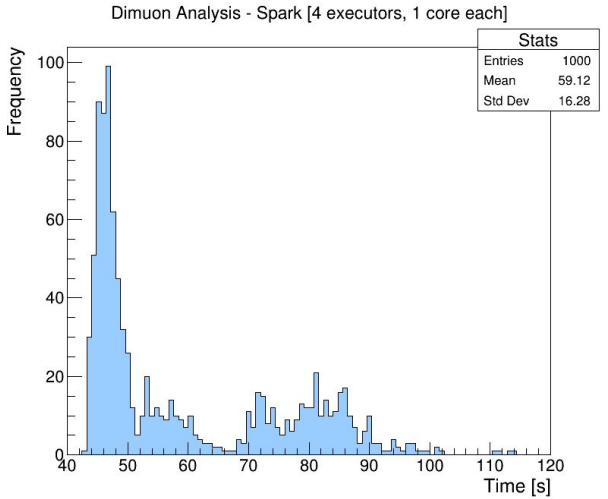
Multithreaded [4 threads]

- 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

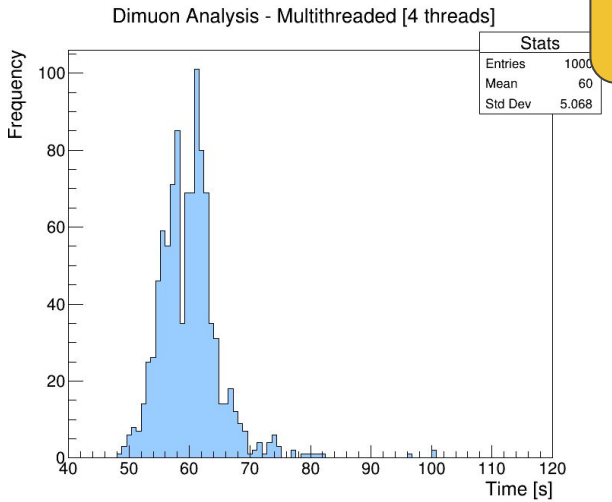
Multithreaded [4 threads]

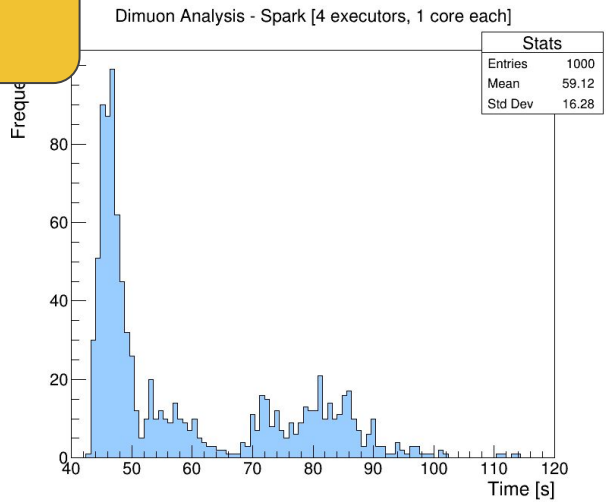
- AVG time [s]: 60.5 +- 12.7
- Most frequent time [s]: 61.2

Spark [4 executors, 1 core each]

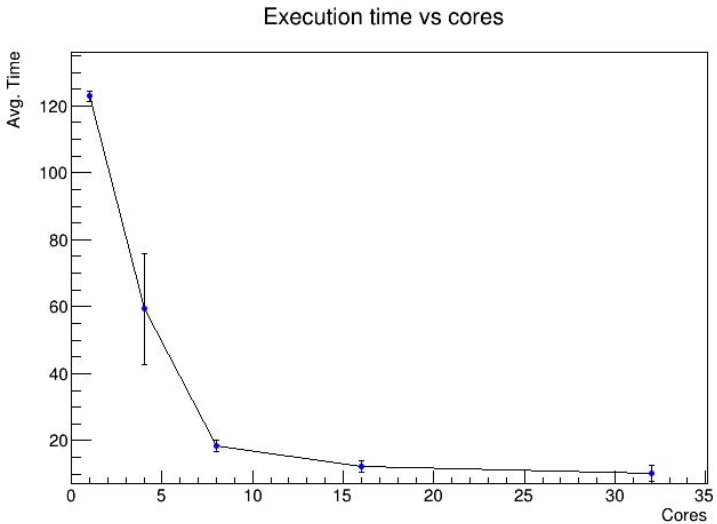
- AVG time [s]: 59.1 +- 32.5
- Most frequent time [s]: 46.8

Most frequent time is much lower with Spark





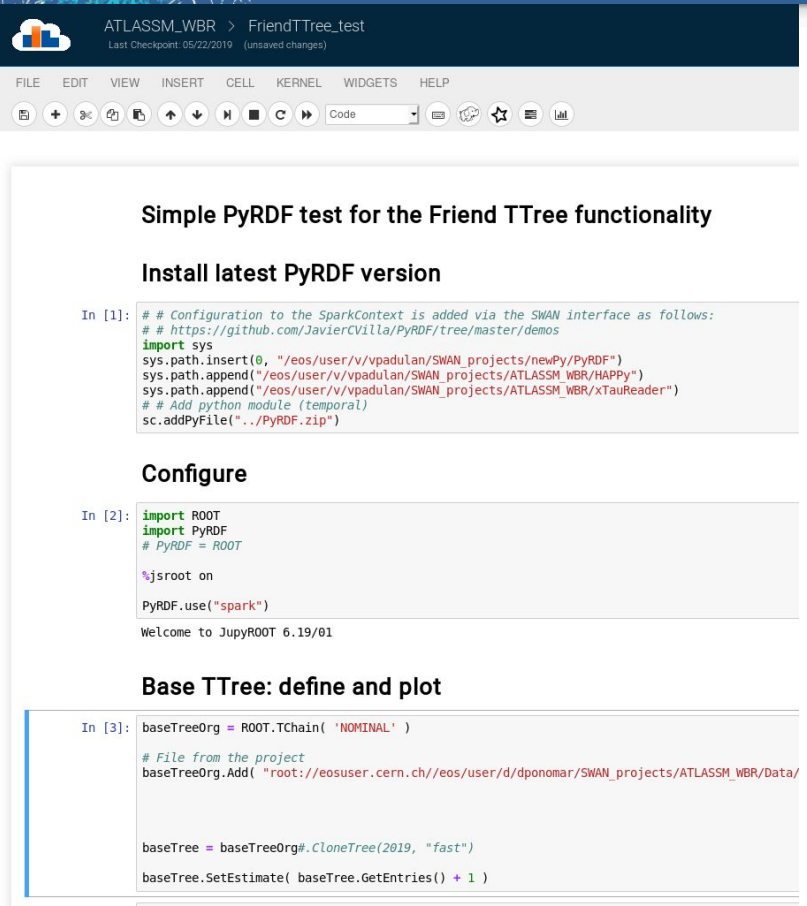
Dimuon Analysis Tests



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

Use cases

ATLAS use case



- ATLAS user with a ROOT RDataFrame based analysis
- Intricated package with multiple dependencies

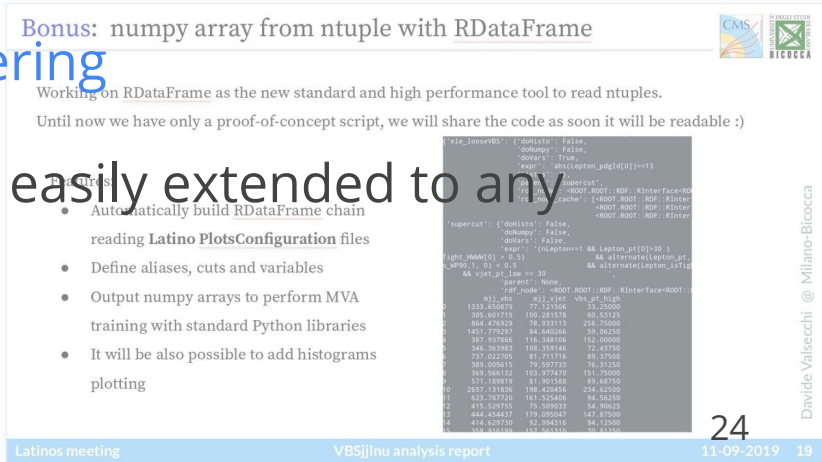
New Feature developed:
Friend trees with a Spark backend!

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

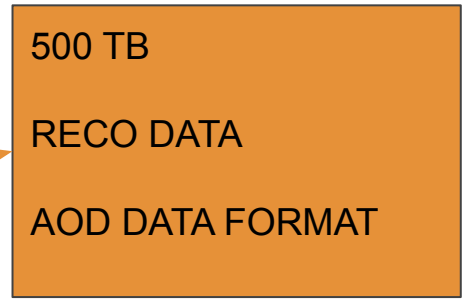
Very early stage, but this approach can be easily extended to any NanoAOD analysis.



preliminary presentation, analysis repo

A challenging use case

Data from the TOTEM experiment



- 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 succesful attempt in the user session) and the Spark executors.

repo


Thank you!

FriendTree Compatibility

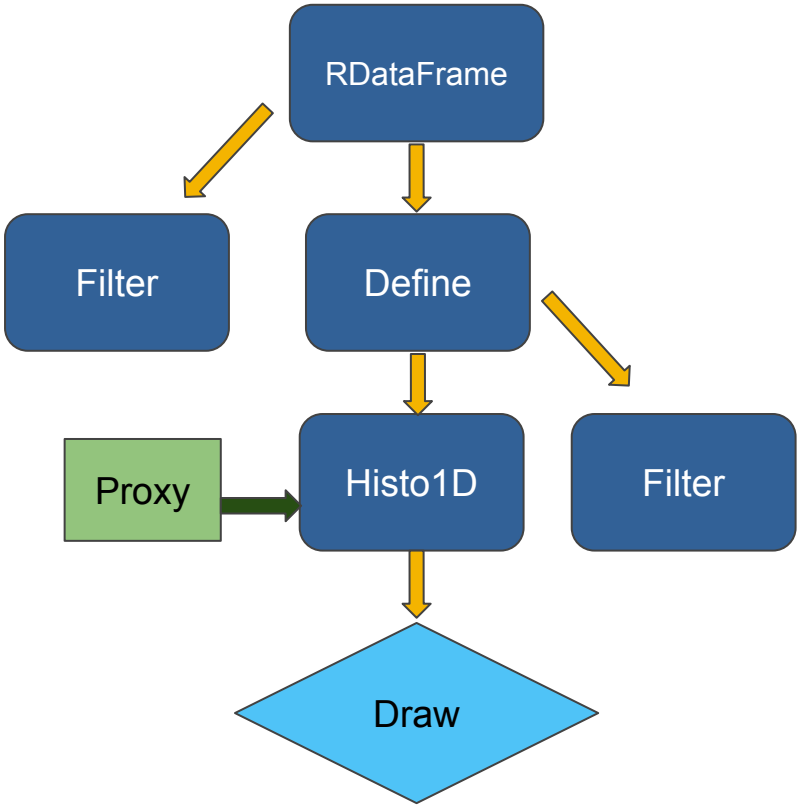
During map phase:

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

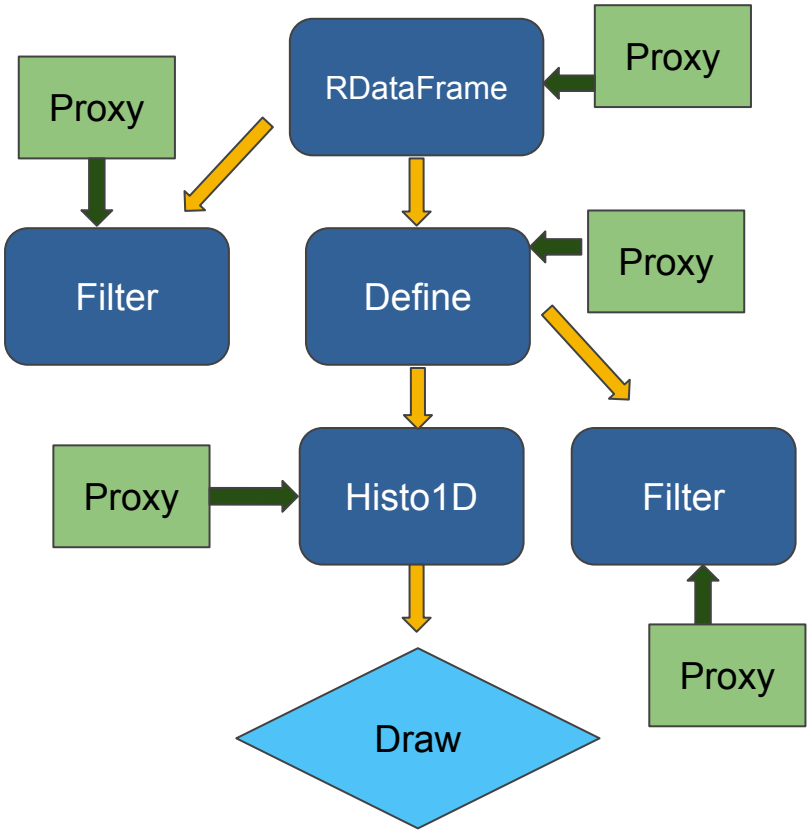


Before release 0.1 :

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