Distributed Data Analysis with ROOT RDataFrame

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

ROOT

Data Analysis Framework

https://root.cern

Introduction

The HEP DataFrame

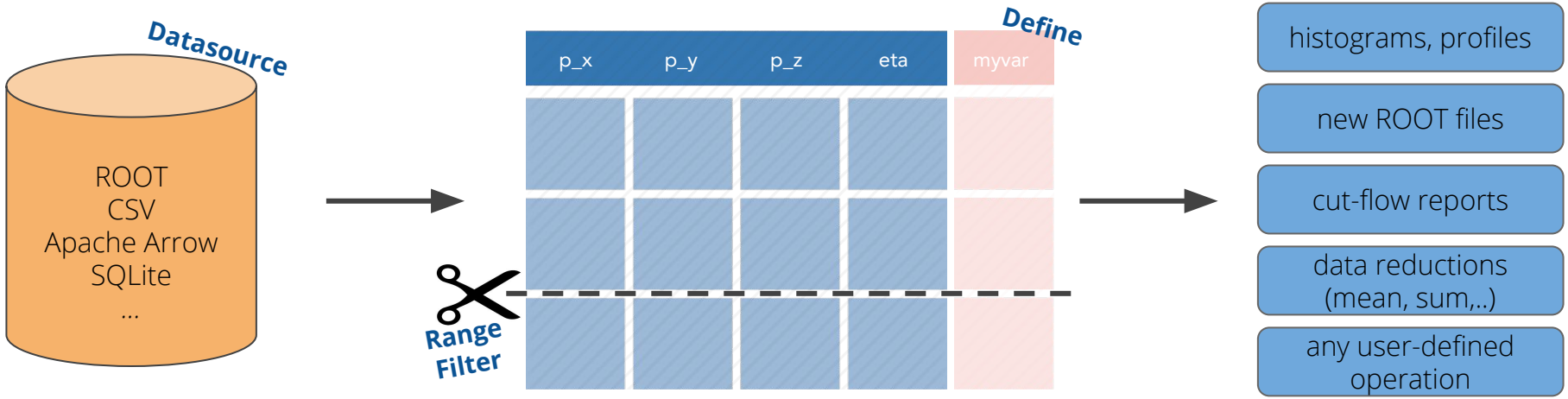
- strive for a **simple programming model** based on declarative programming
- expose modern, elegant interfaces that are **easy to use correctly** and hard to use incorrectly
- **transparently benefit from multi-core** hardware
- make **common tasks simple, complex tasks possible**
- consistent support for HEP languages: **C++ and Python**

RDataFrame, officially part of ROOT since v6.14, tries to incarnate these ideas in the context of HEP analyses and HEP data manipulation

RDF user guide, cheat-sheet

DOI 10.5281/zenodo.260230

RDataFrame in a Nutshell



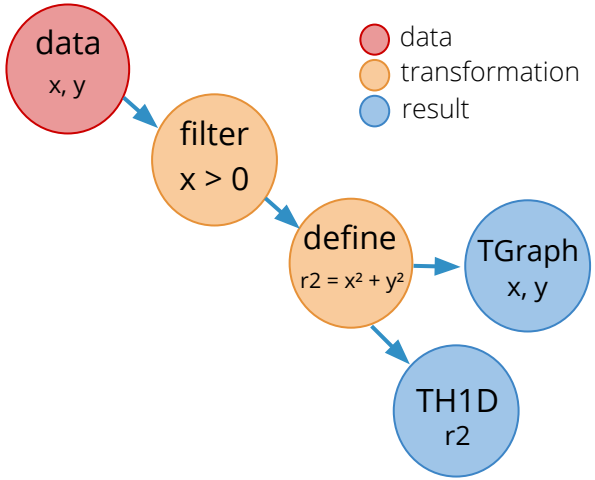
Analysis as Computation Graphs

```
from ROOT import RDataFrame
df  = RDataFrame(dataset);
df2 = df.Filter("x > 0")
        .Define("r2", "x*x + y*y");

rHist = df2.Histo1D("r2");

g = df2.Graph("x","y")
```

Internal computation graph



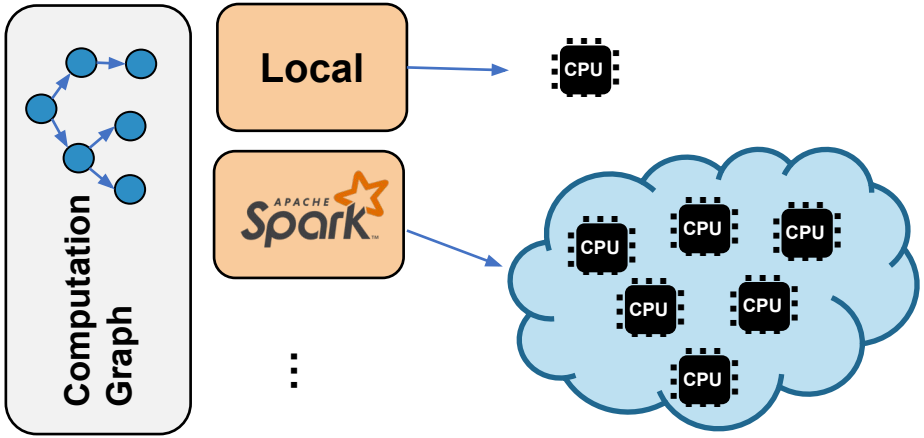
PyRDF: Distributed RDataFrame

- The RDataFrame programming model is implicitly parallel
  - Runs on multi/many core architectures
  - But it can also exploit **distributed infrastructures** !
- **PyRDF**: Python library on top of ROOT RDataFrame
  - Enables distributed execution of RDataFrame workflows
  - Modular design: multiple backends can be plugged in
- **Spark** plugin implemented: submits RDataFrame computations to Spark clusters

Code here!

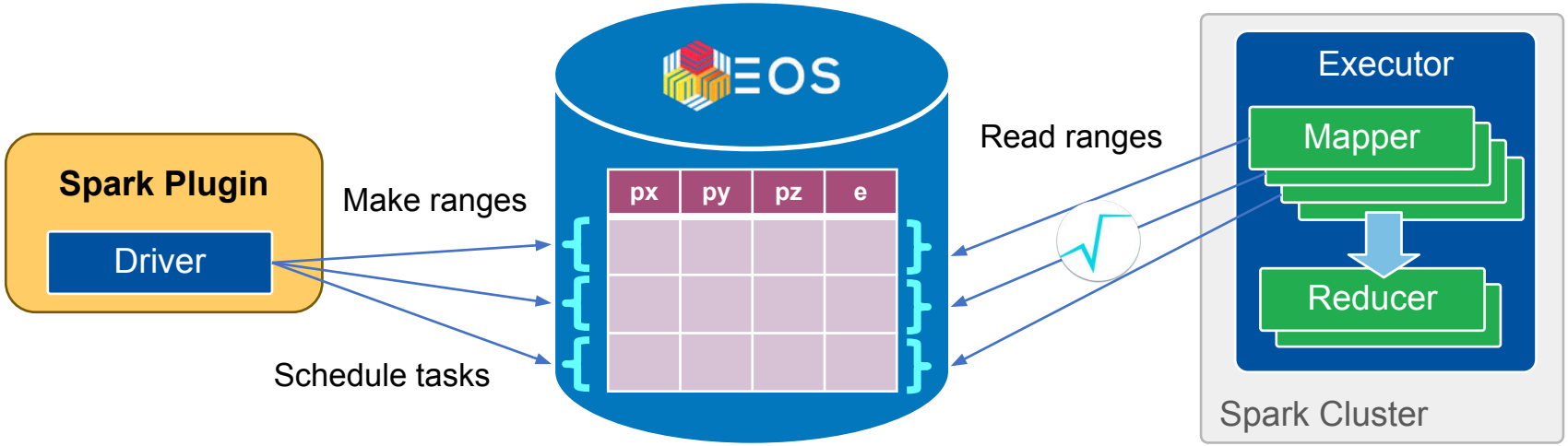
```
d = RDataFrame(dataset)
f = d.Define(...)
     .Define(...)
     .Filter(...)

h1 = f.Histo1D(...)
h2 = f.Histo2D(...)
g   = f.Graph(...)
```



Spark Backend

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



Features Overview

Programming Model

- Minimal changes on user's code

```
from ROOT import RDataFrame

# Initialize RDataFrame object
df = RDataFrame(dataset)

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

# Display histogram
rHist.Draw()
```

Programming Model

- Minimal changes on user's code

```
from ROOT import RDataFrame

# Initialize RDataFrame object
df = RDataFrame(dataset)

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

# Display histogram
rHist.Draw()
```

RDataFrame via PyRDF

```
from PyRDF import RDataFrame

# Initialize RDataFrame object
df = RDataFrame(dataset)

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

# Display histogram
rHist.Draw()
```

API: Backend Selection

- Multi-backend support
  - Dynamic switch of backends

Move to local backend

Spark

Local

```
# Select Spark backend
PyRDF.use("spark")

# Initialize RDataFrame object
df = RDataFrame(dataset)

# Operations run in Spark
df2 = df.Filter("x > 0")
         .Define("r2", "x*x + y*y")
rHist = df2.Histo1D("r2")
# Trigger event loop
sd = rHist.GetStdDev()

# Switch back to Local backend
PyRDF.use("local")

# Operations run locally
df3 = df2.Filter("r2 % 2 == 0")
```

API: C++ Headers and Libraries

- Include C++ headers and libraries
  - PyRDF makes them available in the distributed nodes

myfunc.hxx

```
bool myfunc(int a, int b);
```

myfunc.cxx

```
bool myfunc(int a, int b) {
    return a < b;
}
```

```
# Add analysis headers and libraries
PyRDF.include_headers("myfunc.hxx")
PyRDF.include_shared_libraries("myfunc.so")

# Initialize RDataFrame object
df = RDataFrame(dataset)

# Operations run in distributed backend
df2 = df.Define("res", "myfunc(x,y)")
```

Calls from JITted code

RDataFrame Snapshot

- RDataFrame Snapshot allows to save data to a file

```
new_df = df.Filter("x > 0")
           .Define("z", "sqrt(x*x + y*y)")
           .Snapshot("tree", "newfile.root")
```

We filter the data, add a new column, and then save everything to file

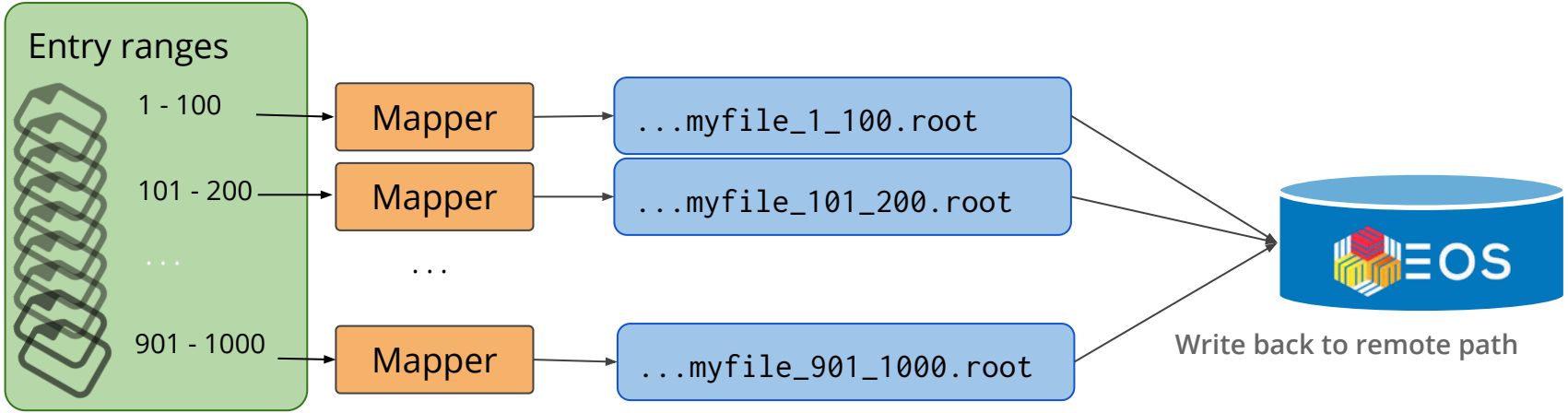
Distributed Snapshot

```
import PyRDF
PyRDF.use("spark")

# RDF Operations ...

new_df = df.Snapshot(remotepath)
```

Path to a remote file:
`root://eosuser.cern.ch//mypath/myfile.root`

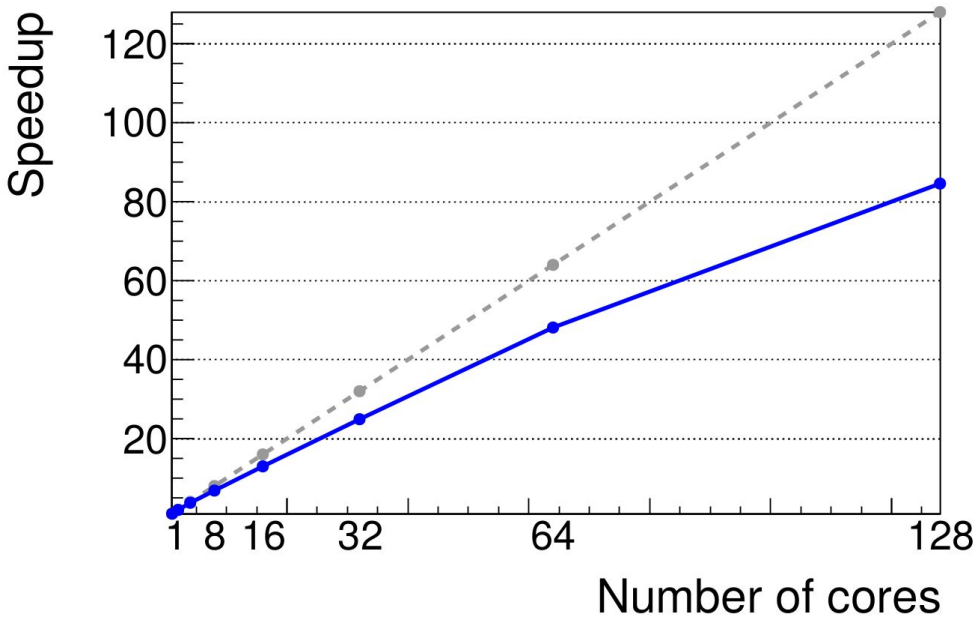


Use Case

Real Example: TOTEM Analysis

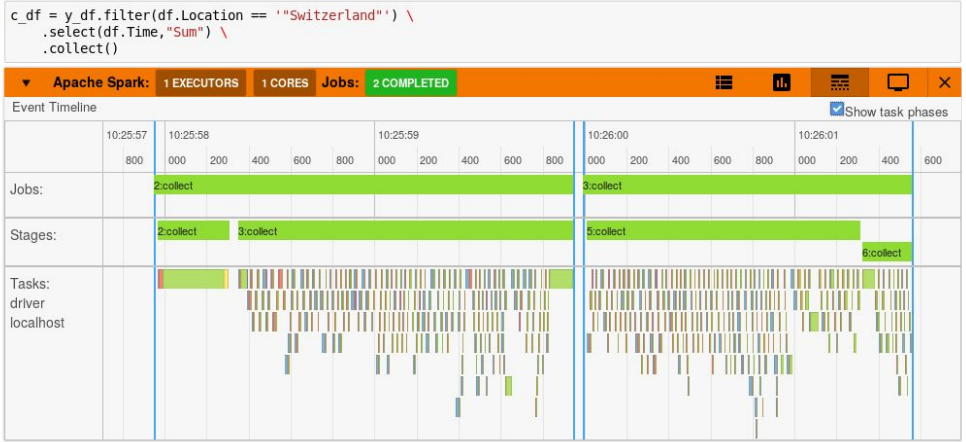
- TOTEM Experiment analysis coded with RDataFrame
  - Proton-proton elastic scattering
- **Spark** backend
- **4.7TB** input dataset on EOS
- **Get to physics results faster!**
  - From 13 hours to 10 minutes
- Launched from **SWAN** to a dedicated **Spark cluster**

Link to Euro-Par 2019 paper



Distributed Monitoring

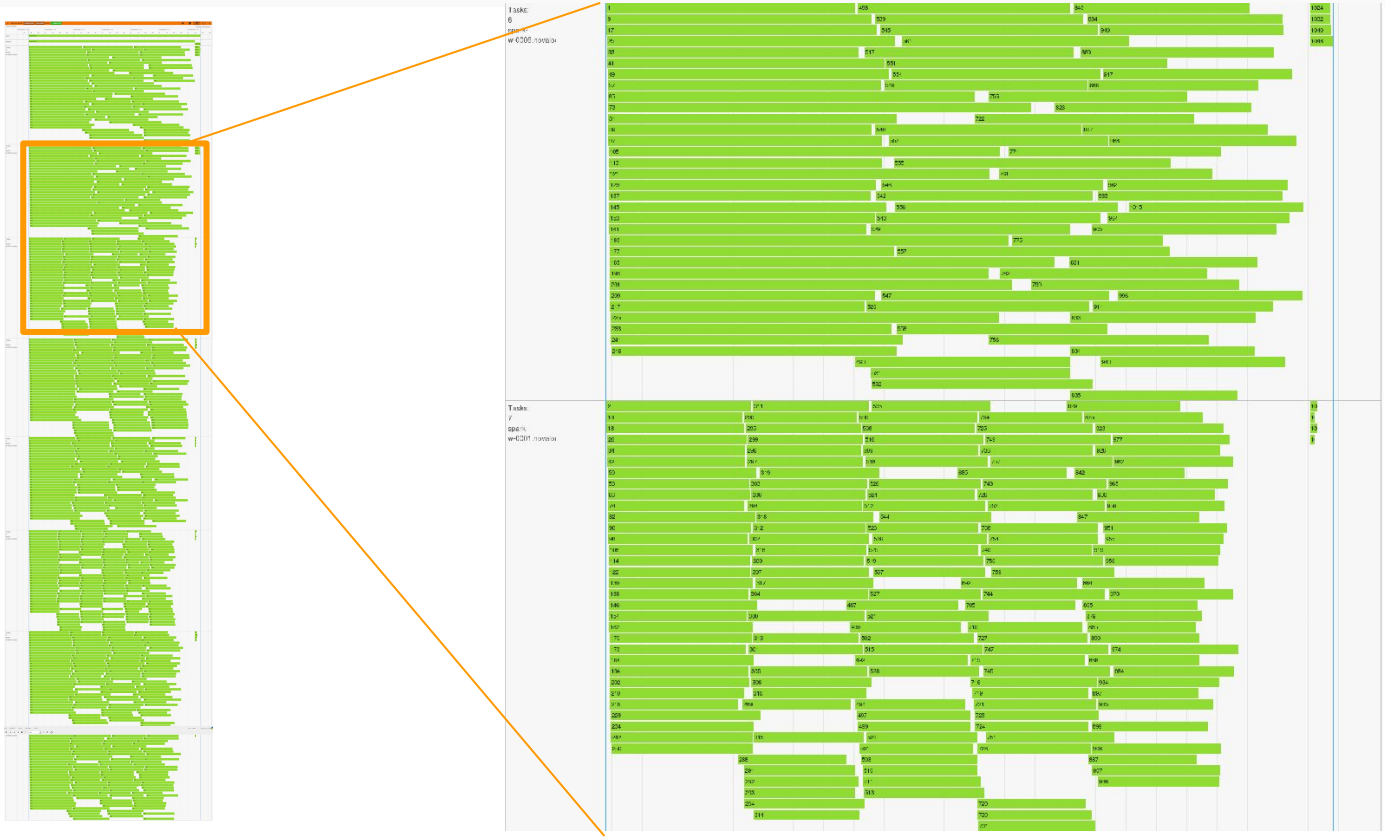
- **Bridge the gap** between interactive computing and distributed data processing
- Automatically appears when a Spark job is submitted from a cell
- Progress bars, task timeline, resource utilisation



Google Summer of Code

Code here!

Useful for Debugging



Conclusions

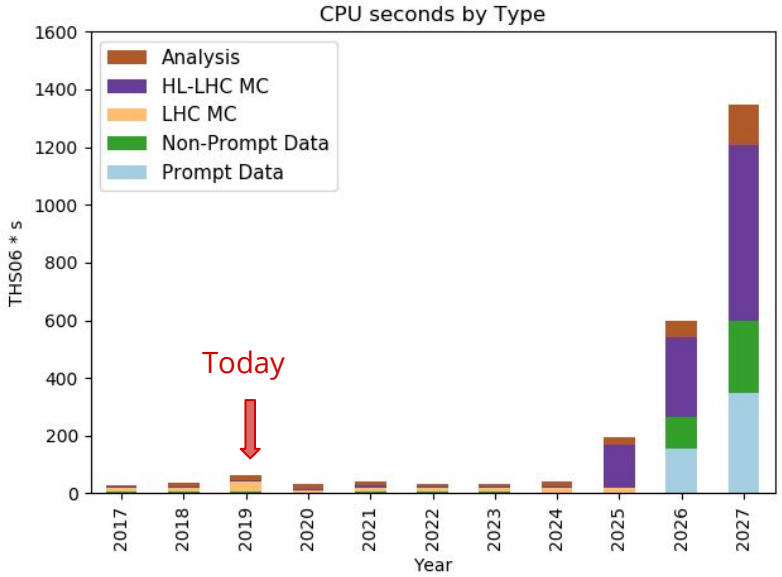
- The increase in physics data volumes and complexity is **challenging software** in HEP
  - Adoption of Spark and other Big Data technologies still in its early stages
- Adopting new programming paradigms takes time
  - Declarative analysis
- RDataFrame and **PyRDF** try to combine:
  - Easy to use programming model
  - Implicit parallelization (local, distributed)
- **To try it out**: PyRDF is distributed with LCG releases (from LCG 96)
  - SWAN provides additional goodies: easy connect to Spark clusters, live monitoring

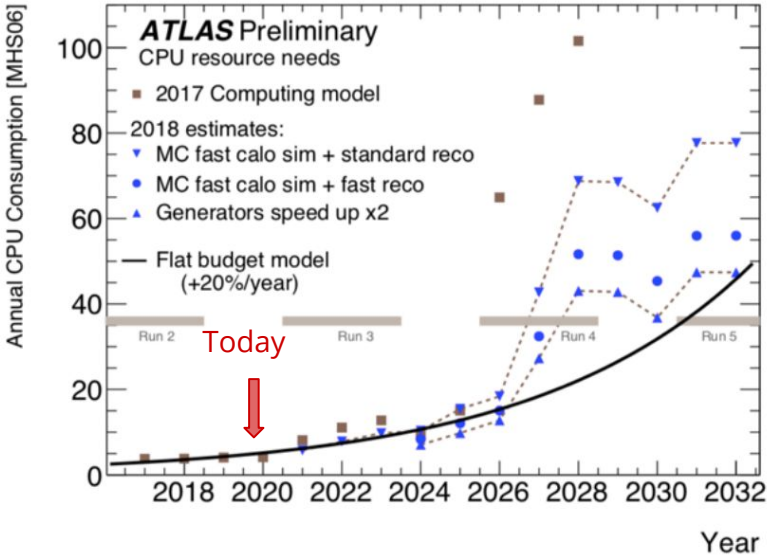
Thank you!

Backup

Reasons to run distributedly

- The amount of data processed by HEP scientists is going to increase drastically

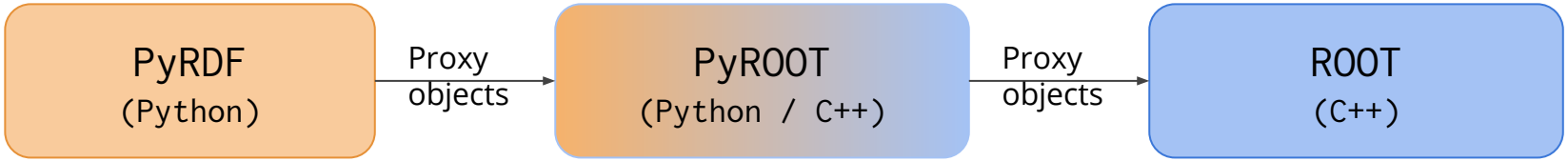




sources: CMS, ATLAS

PyRDF: Main design principles

- Delay computations as much as possible
- Avoid data format conversion
- Change the backend dynamically
- Minimal changes on user's code
  - Changing the mindset of programmers takes time
  - Keep the same interface offered by RDataFrame in Python
  - Support *local* as a backend



Backend Configuration

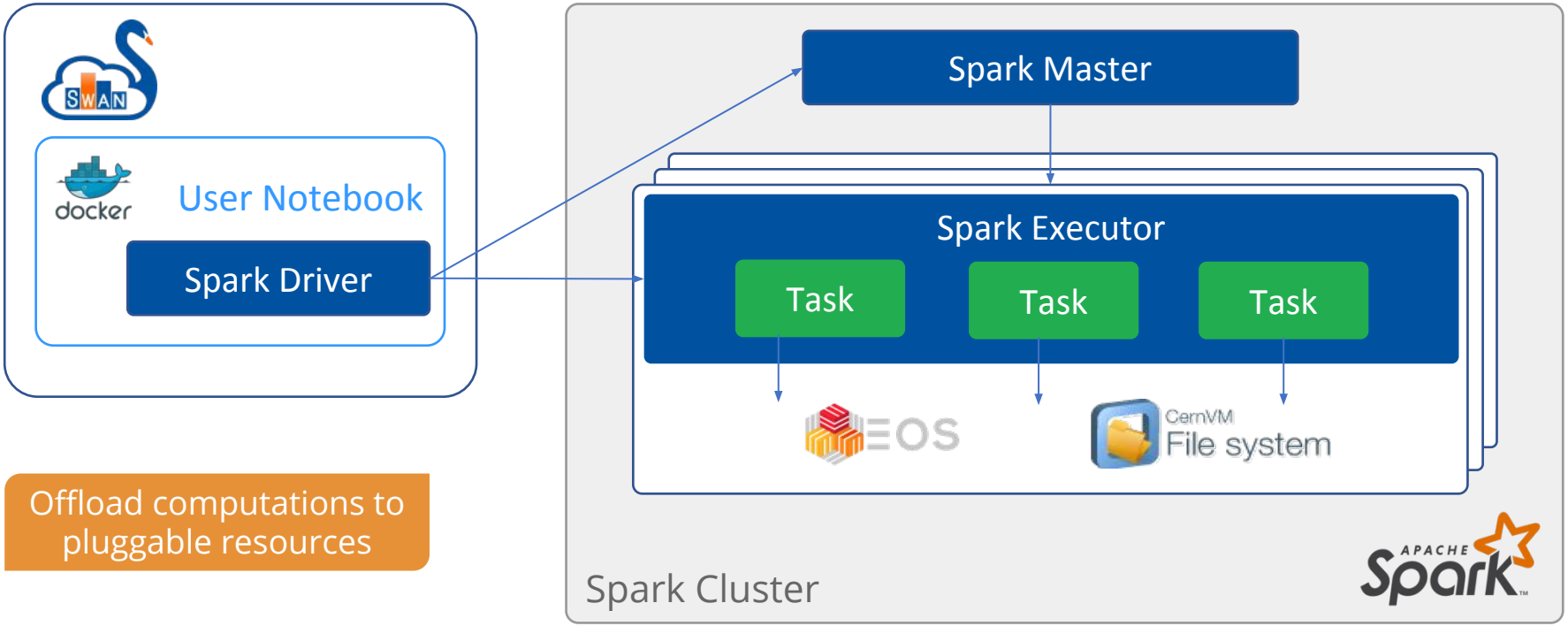
- Entrypoint to backend configuration
  - Explicit parameters
  - Accept all backend parameters

```
import PyRDF

# Configure Spark backend
PyRDF.use("spark", {
            "npartition": 4,
             "spark.executor.instances": 5
           })

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

Integration with Spark



Offload computations to pluggable resources

Spark Connector

Spark > physics_analysis_using_swan_spark_template
(autosaved)

FILE EDIT VIEW INSERT CELL KERNEL HELP Not Trusted Python 2

Markdown

Integration of SWAN with Spark clusters

This notebook demonstrates the functionality provided by a SWAN prototype machine that allows to offload computations to an external Spark cluster. The Spark version we are going to use is 2.1.0 and we are going to connect to the analytix cluster (as previously selected in the SWAN web form).

Step 1 - Acquire the necessary credentials to access the Spark cluster.

In [1]:
```
import getpass
import os, sys, re

print("Please enter your password")
ret = os.system("echo \"%s\" | kinit" % re.escape(getpass.getpass()))

if ret == 0: print("Credentials created successfully")
else:        sys.stderr.write('Error creating credentials, return code: %s\n' % ret)
```

Spark > Spark_Simple
(autosaved)

FILE EDIT VIEW INSERT CELL KERNEL HELP

Markdown

Simple example with Spark

This notebook illustrates the use of Spark in SWAN.

The current setup allows to execute PySpark operations on a loca
small datasets.

In the future, SWAN users will be able to attach external Spark clu
Moreover, a Scala Jupyter kernel will be added to use Spark from

Import the necessary modules

Spark clusters connection

You are going to connect to:
hadalytic

You can configure the following options.
Environment variables can be used via {ENV_VAR_NAME}.

Add a new option

Write the option name...

Bundled configurations

Include NXCALS options

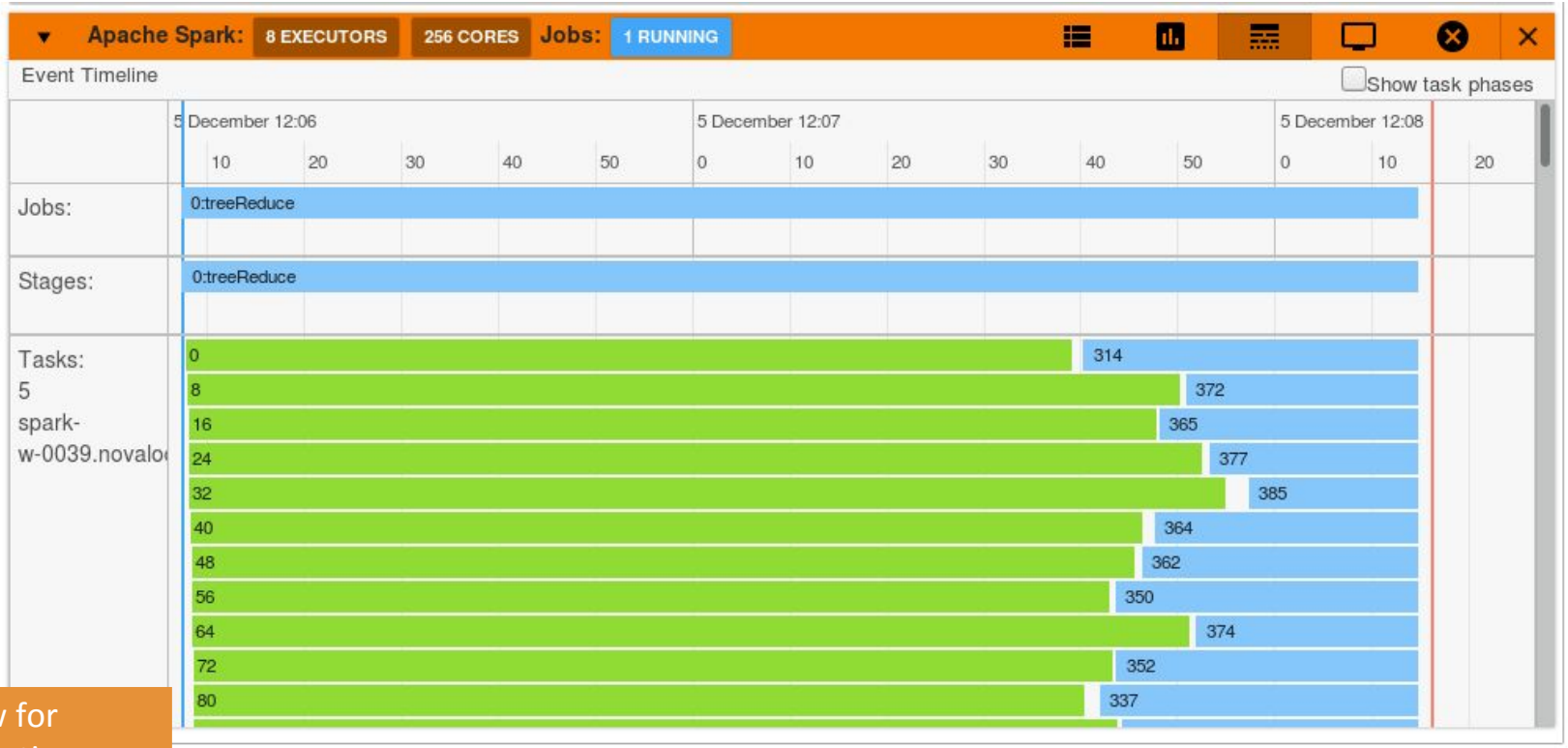
Selected configuration

- **spark.shuffle.service.enabled**
  false
- **spark.driver.memory**
  2g
- **spark.executor.instances**
  4

Connect

Configure Spark and connect to cluster with a click

Spark Monitor



Allow for optimizations

Spark Monitor: Debugging

- Easy to spot inefficiency situations
  - Optimize use of resources (cores)