

Distributed Data Analysis with ROOT RDataFrame

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

ROOT

Data Analysis Framework

<https://root.cern>



Introduction



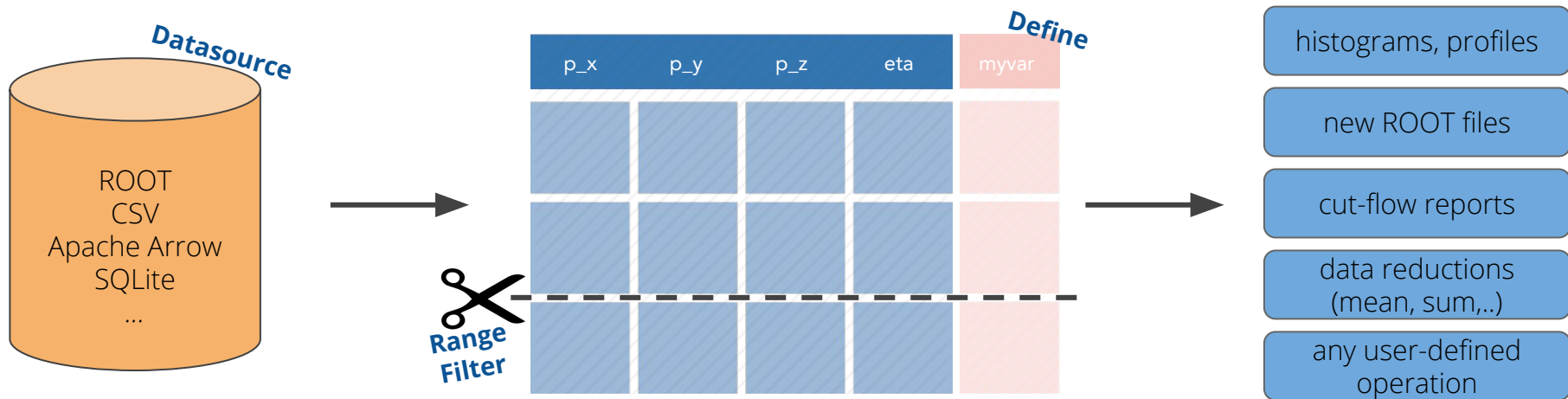
The HEP DataFrame

- ▶ 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



RDataFrame in a Nutshell

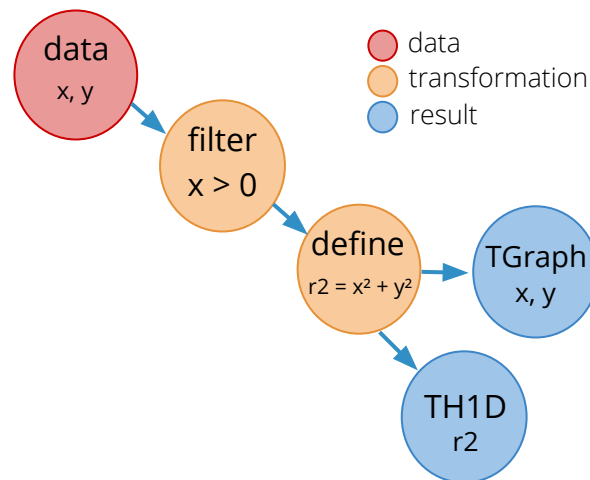




Analysis as Computation Graphs

```
import ROOT
df  = ROOT.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



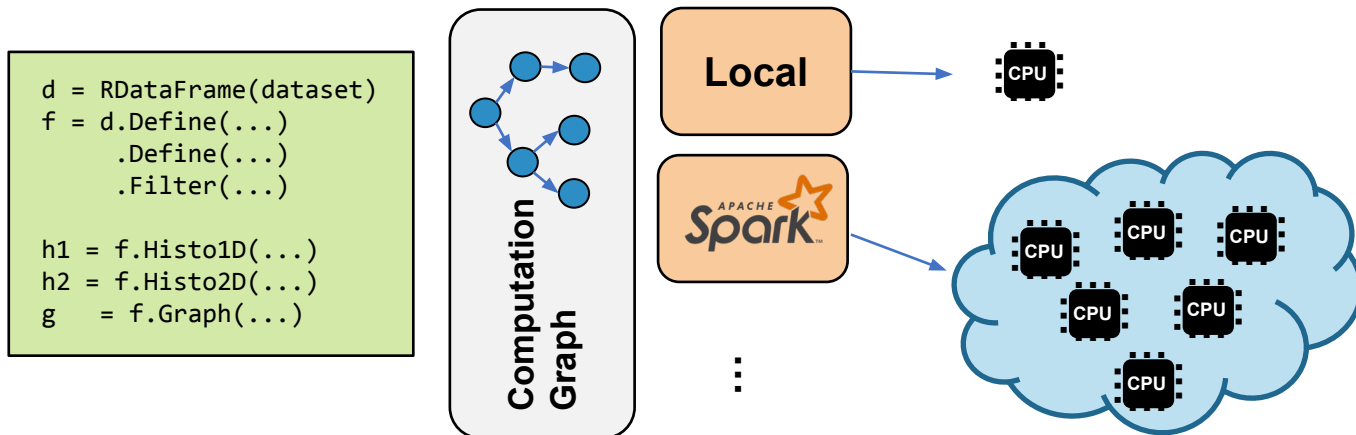


PyRDF: Distributed RDataFrame

- ▶ 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** backend implemented: submits RDataFrame computations to Spark clusters

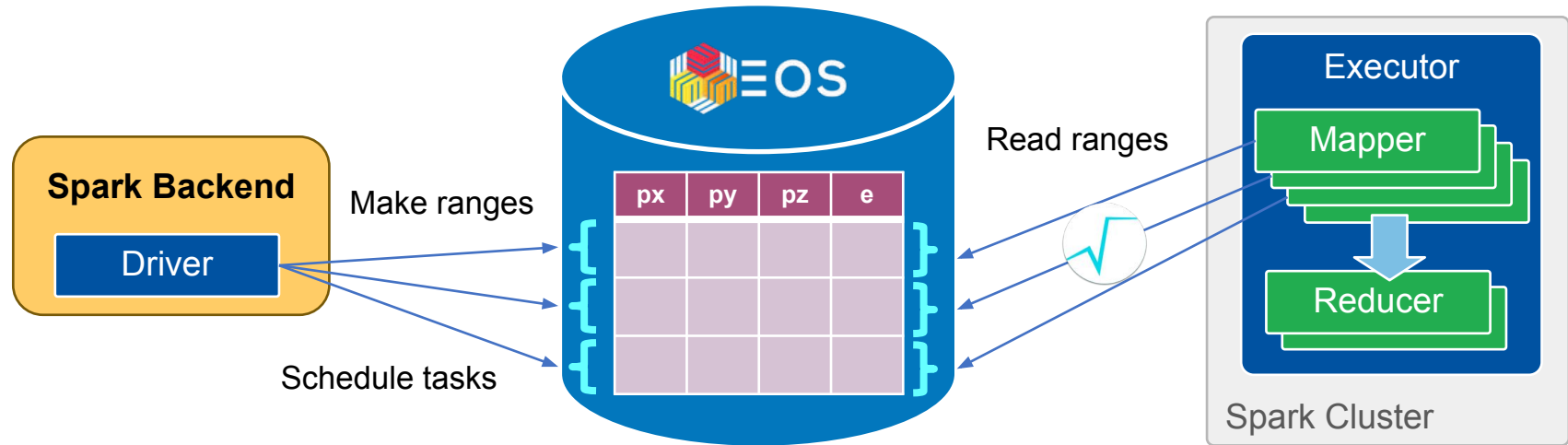


[Code here!](#)



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



- ▶ Minimal changes on user's code

```
import ROOT

# Initialize RDataFrame object
df = ROOT.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

RDataFrame via PyRDF

```
import ROOT

# Initialize RDataFrame object
df = ROOT.RDataFrame(dataset)

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

# Display histogram
rHist.Draw()
```

```
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()
```



API: Backend Selection

- ▶ Multi-backend support
 - Dynamic switch of backends

Move to local
backend



Spark

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

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

Local

```
# 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 = PyRDF.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

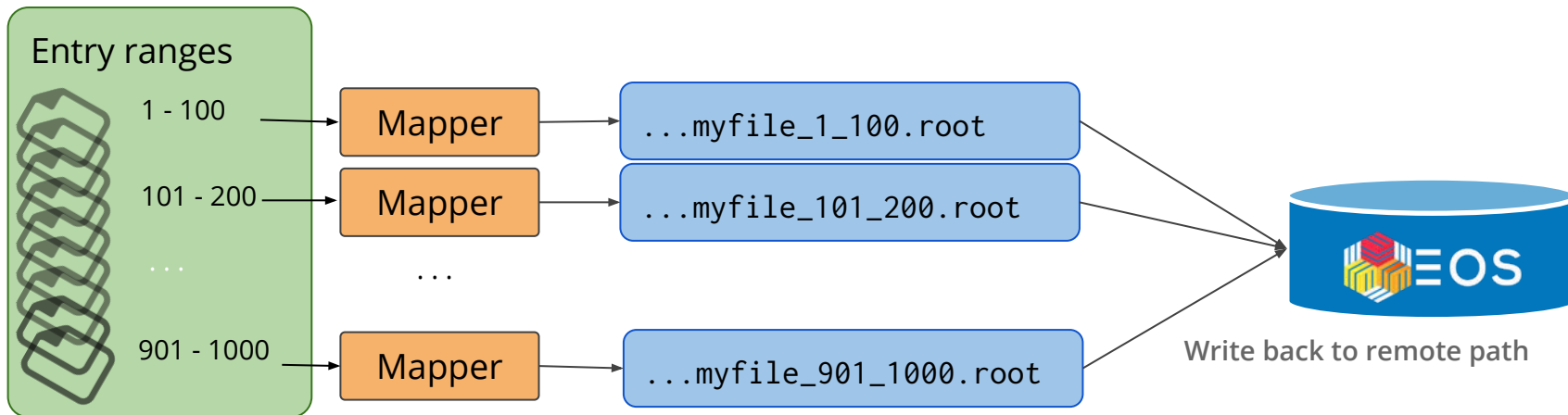
Distributed Snapshot

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

# RDF Operations ...

new_df = df.Snapshot(remotepath)
```

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



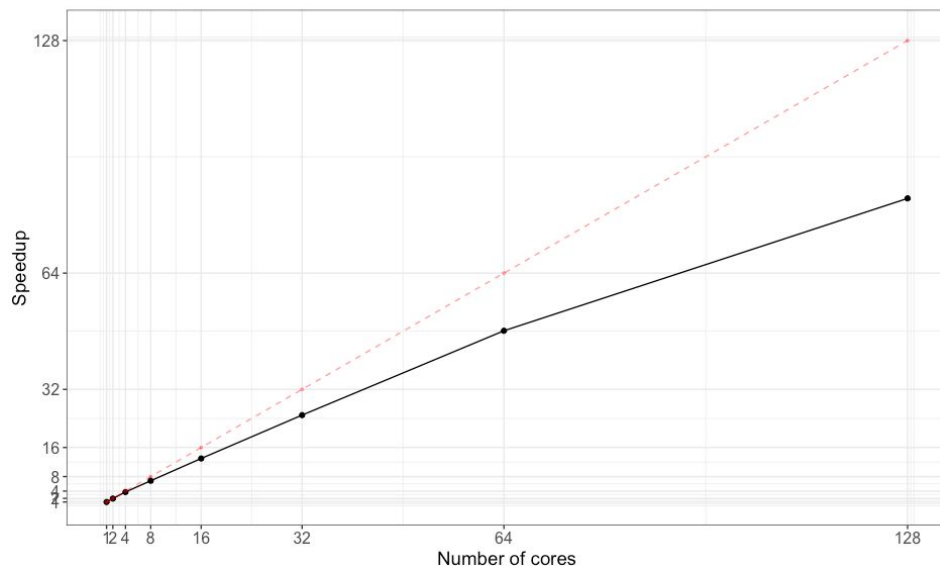


Use Case



Real Example: TOTEM Analysis

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





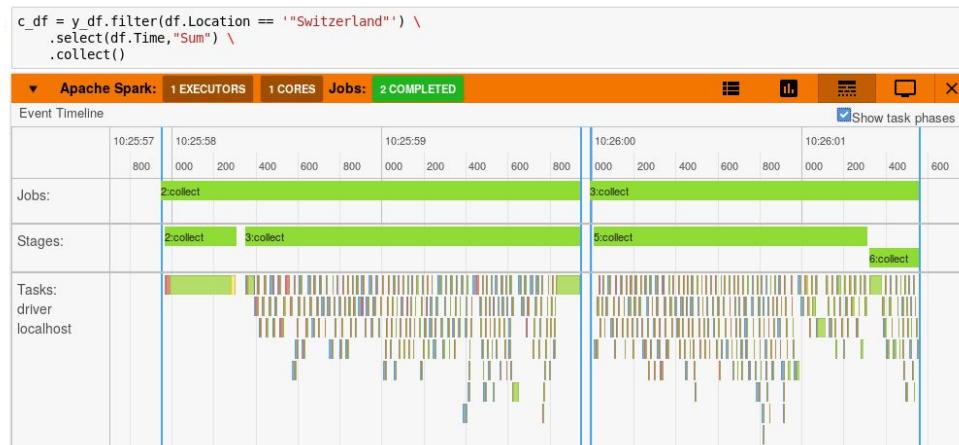
Distributed Monitoring

- ▶ **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!](#)



Apache Spark: 1 EXECUTORS 4 CORES Jobs: 2 COMPLETED

Job ID	Job Name	Status	Stages	Tasks	Submission Time	Duration
2	reduce	COMPLETED	2/2	48 / 48	5 minutes ago	3s
Stage details for Job 2:						
Stage Id	Stage Name	Status	Tasks	Submission Time	Duration	
5	reduce	COMPLETED	32 / 32	5 minutes ago	2s	
4	coalesce	COMPLETED	16 / 16	5 minutes ago	0s	
3	foreach	COMPLETED	1/1 (1 skipped)	32 / 32	5 minutes ago	1m:20s
Stage details for Job 3:						
Stage Id	Stage Name	Status	Tasks	Submission Time	Duration	
6	coalesce	SKIPPED	0 / 16	Unknown	-	
7	foreach	COMPLETED	32 / 32	5 minutes ago	1m:20s	



Useful for Debugging





Conclusions

- ▶ The increase in physics data volumes and complexity is **pushing software** at CERN
 - Adoption of Spark and other big data technologies still in its early stages
- ▶ Adopting new programming paradigms takes time
 - Declarative analysis
 - Pushing computations to data
- ▶ RDataFrame and **PyRDF** try to combine:
 - Easy to use programming model
 - Implicit parallelization (local, distributed)



Thank you!

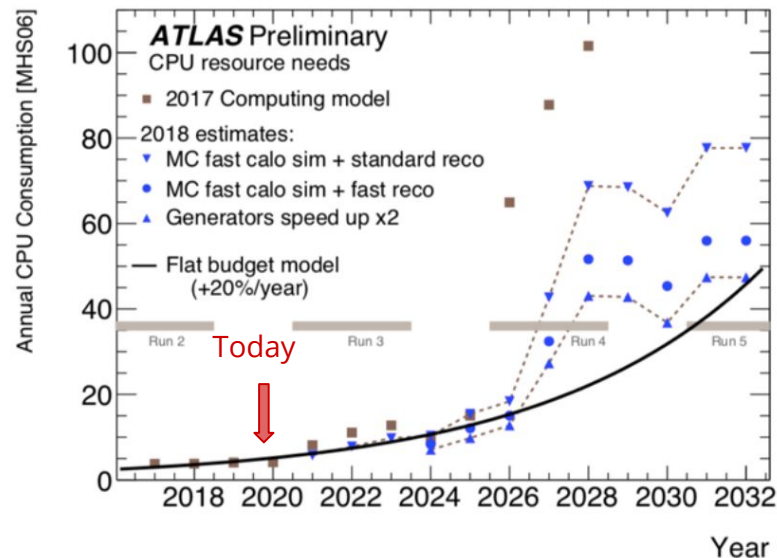
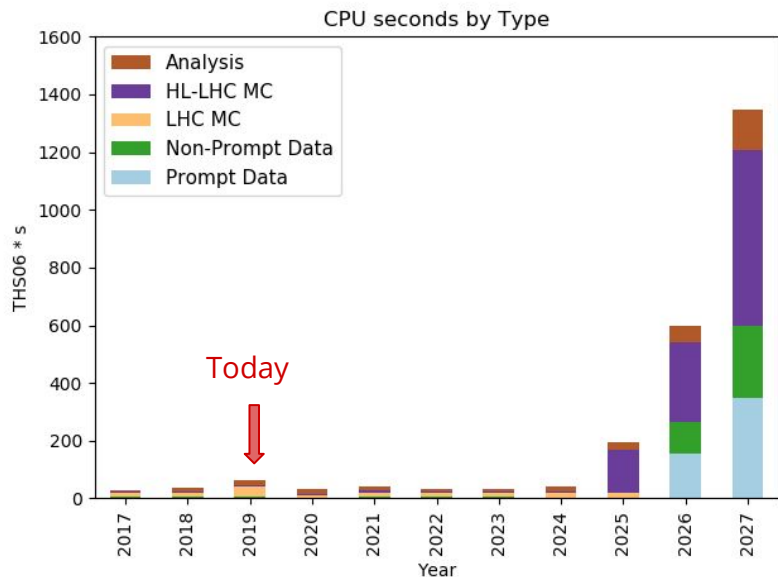


Backup



Reasons to run distributedly

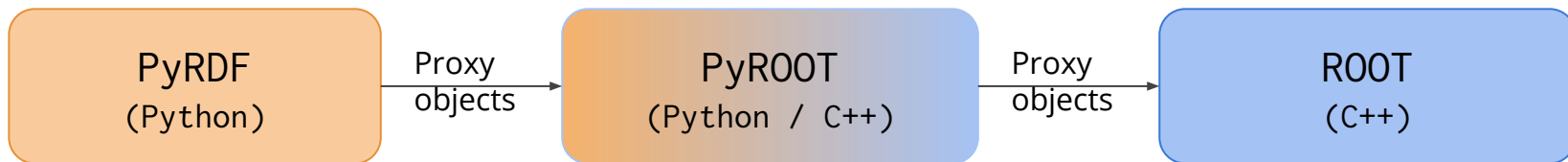
- ▶ The amount of data processed by HEP scientists is going to increase drastically





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)
```




The SWAN Service

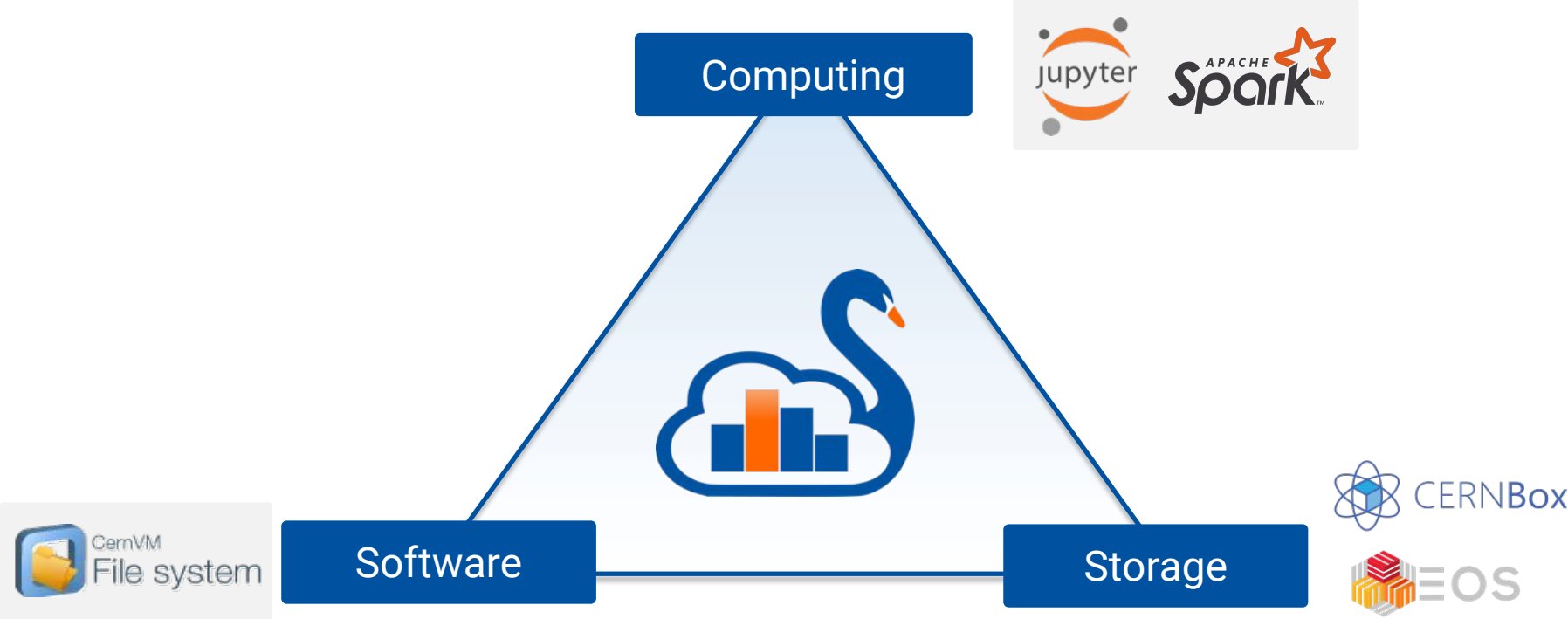
- ▶ **SWAN**: Service for Web-based Analysis
- ▶ **Interactive computing** platform for scientists
 - Based on Jupyter notebooks
- ▶ Analysis with only a web browser
- ▶ Easy **sharing of results**
- ▶ Integrated with CERN resources
 - Storage, software and computing



<https://swan.web.cern.ch>



SWAN Pillars



SWAN Interface: Notebooks

Simple_ROOTbook_cpp.ipynb
(view only)

Simple ROOTbook (C++)

This simple ROOTbook shows how to create a [histogram](#), [fill it](#) and [draw it](#). The language chosen is C++.

In order to activate the interactive visualisation we can use the `JSROOT` magic:

```
In [1]: %jsroot on
```

Now we will create a [histogram](#) specifying its title and axes titles:

```
In [2]: TH1F h("myHisto", "My Histo;X axis;Y axis", 64, -4, 4)
(TH1F *) Name: myHisto Title: My Histo NbinsX: 64
```

If you are wondering what this output represents, it is what we call a "printed value". The ROOT interpreter can indeed be instructed to "print" according to certain rules instances of a particular class.

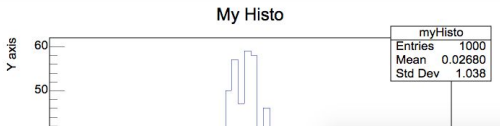
Time to create a random generator and fill our histogram:

```
In [3]: TRandom3 rndmGenerator;
for (auto i : ROOT::TSeqI(1000)){
  auto rndm = rndmGenerator.Gaus();
  h.Fill(rndm);
}
```

We can now draw the histogram. We will at first create a [canvas](#), the entity which in ROOT holds graphics primitives.

```
In [4]: TCanvas c;
```

```
In [5]: h.Draw();
c.Draw();
```



NAME	STATUS	MODIFIED
Proj1	🔗	5 days ago
Proj2		15 days ago
Project		21 days ago
Project 1		2 months ago
Project 2		4 months ago
ProjTest		15 days ago
Spark		7 days ago
SWAN-Spark_NXCALS_Example		20 days ago
teste		19 days ago


Projects Share CERNBox

SWAN > My Projects

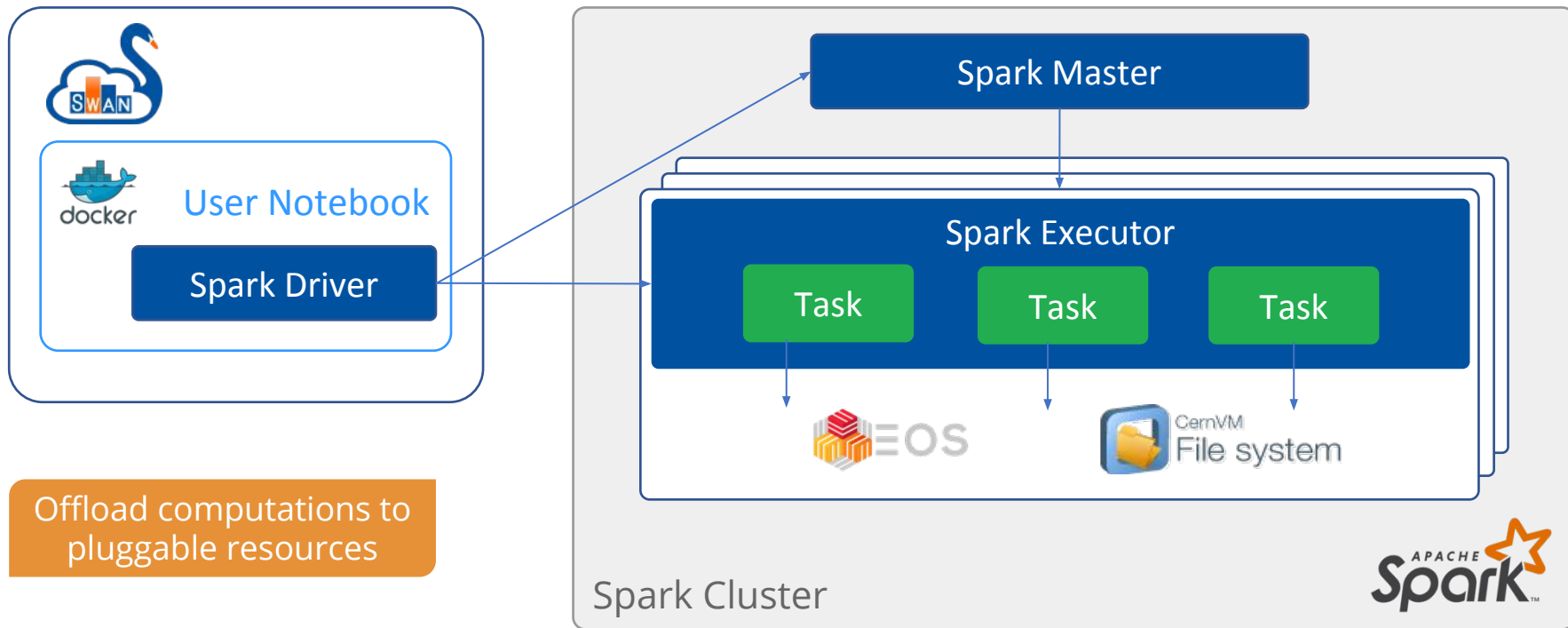
My Projects

NAME	STATUS	MODIFIED
Proj1	🔗	5 days ago
Proj2		15 days ago
Project		21 days ago
Project 1		2 months ago
Project 2		4 months ago
ProjTest		15 days ago
Spark		7 days ago
SWAN-Spark_NXCALS_Example		20 days ago
teste		19 days ago

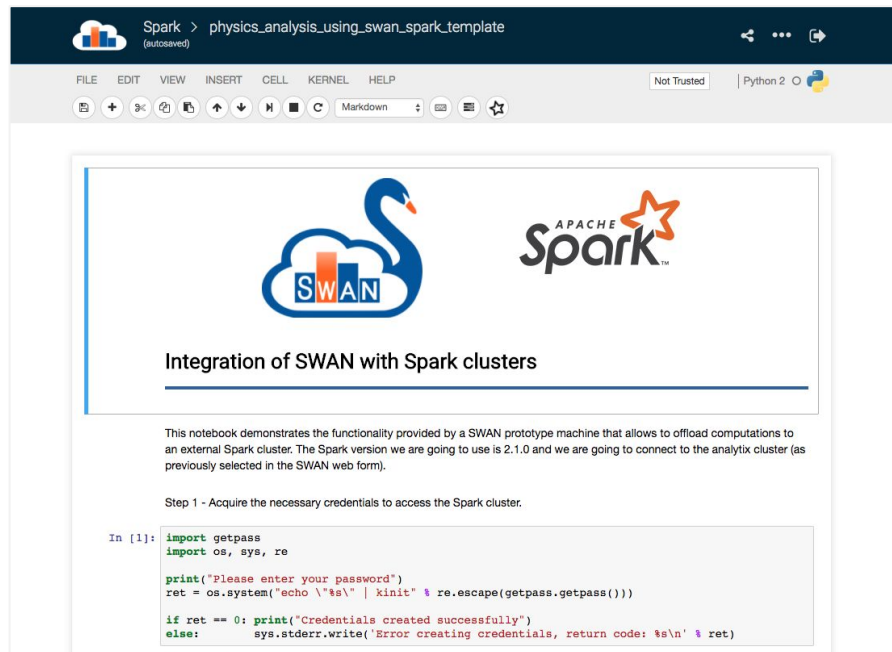
SWAN © Copyright CERN 2017. All rights reserved.
Home | Contacts | Support | Report a bug | Imprint



Integration with Spark



Spark Connector



Spark > physics_analysis_using_swan_spark_template (autosaved)

FILE EDIT VIEW INSERT CELL KERNEL HELP Not Trusted Python 2

Integration of SWAN with Spark clusters

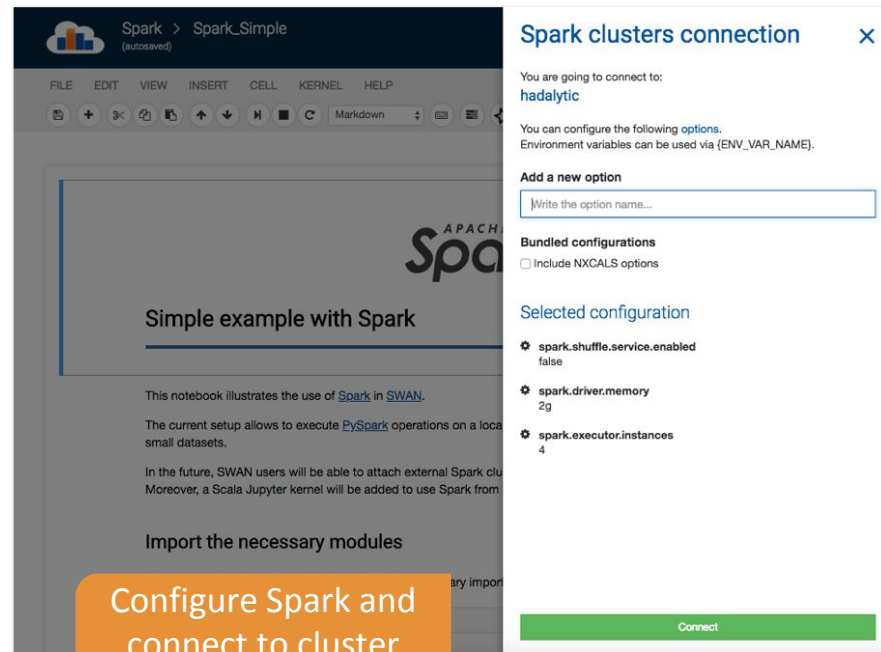
This notebook demonstrates the functionality provided by a SWAN prototype machine that allows to offload computations to an external Spark cluster. The Spark version we are going to use is 2.1.0 and we are going to connect to the analytix cluster (as previously selected in the SWAN web form).

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

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

Bundled configurations

Include NXCALS options

Selected configuration

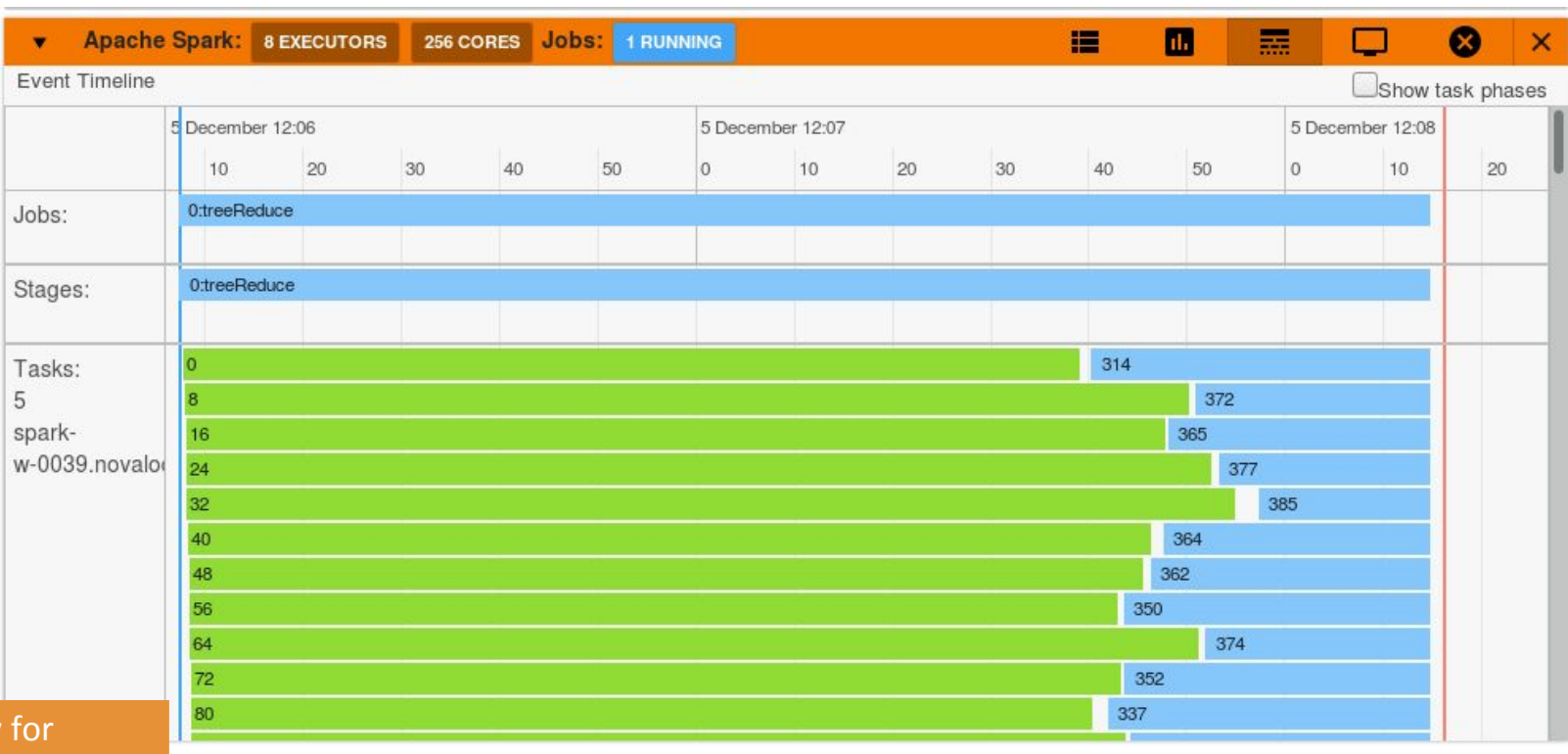
- spark.shuffle.service.enabled: false
- spark.driver.memory: 2g
- spark.executor.instances: 4

Connect

Configure Spark and connect to cluster with a click



Spark Monitor



Allow for
optimizations

Useful for Debugging



- ▶ Easy to spot sources of inefficiencies
 - Optimize use of resources (cores)
- ▶ Led to a better way to manage the task ranges
 - ROOT I/O: prefetch/cache only within task ranges
 - Parallelize generation of ranges