

PyRDF backends

Improving cluster connection interface

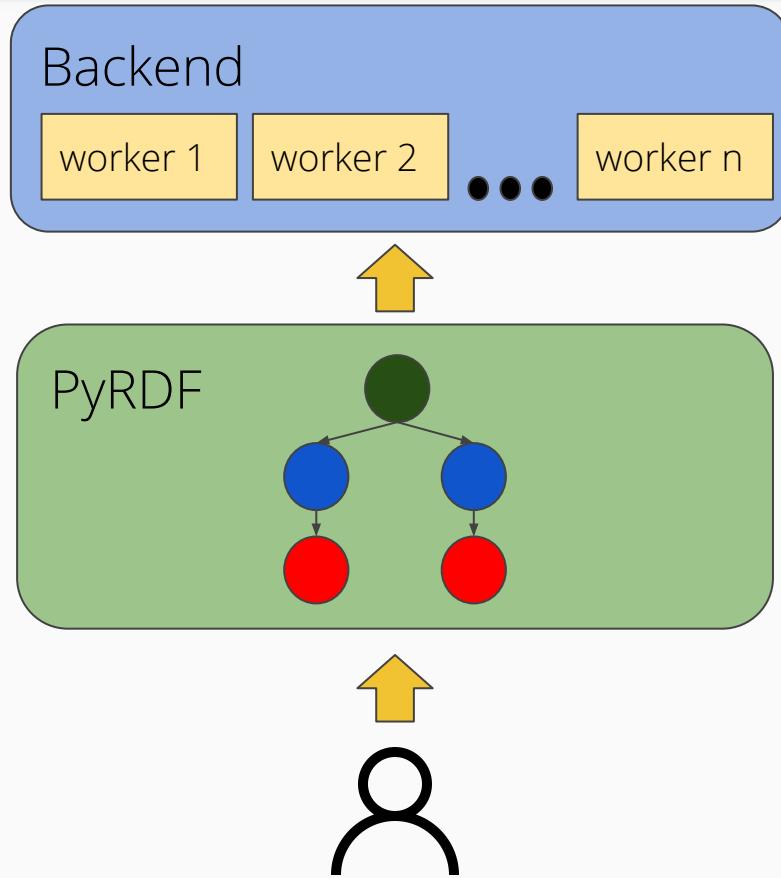
V. E. Padulano



<https://root.cern>

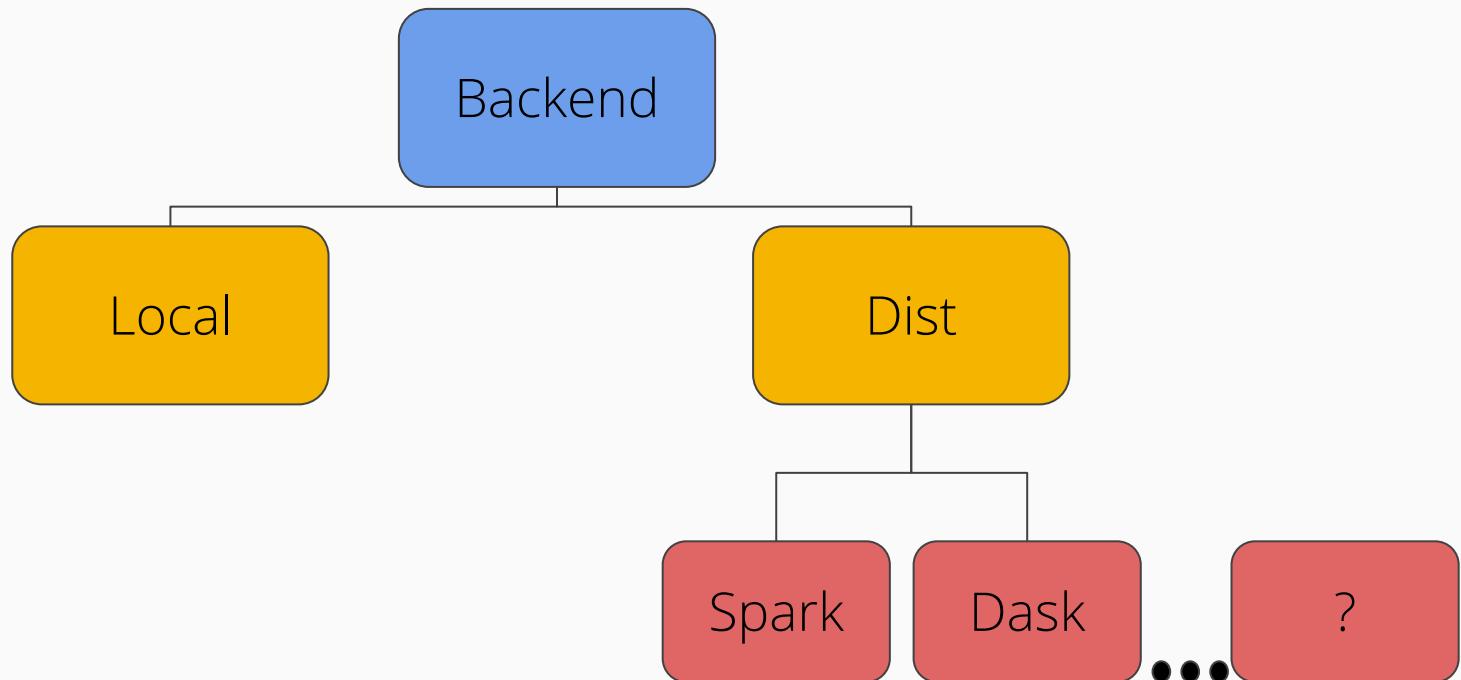


PyRDF distribution model





The backend classes





The connection step

Currently user sets up connection to the distributed cluster through PyRDF .use():

```
import PyRDF

PyRDF.use("spark", conf_dict)

df = PyRDF.RDataFrame(1000)
df.Define(...).Histo1D(...).Draw()
```

```
conf_dict = {
    # Provide Spark options
    'spark.master': 'hostname:port',
    'spark.executor.instances': 1,
    # But also PyRDF options
    'npartitions': 5,
    'use_caching': True
}
```



The connection step

Currently user sets up connection to the distributed cluster through

```
import
```

```
PyRDF.use("spark", conf_dict)
```

```
df = PyRDF.RDataFrame(1000)
```

```
df.Define(...).Histo1D(...).Draw()
```

Uses a string for the name of the backend

```
conf_dict = {  
    # Provide Spark options  
    'spark.master': 'hostname:port',  
    'spark.executor.instances': 1,  
    # But also PyRDF options  
    'npartitions': 5,  
    'use_caching': True  
}
```



The connection step

Currently user sets up connection to the distributed cluster through

```
import
```

```
PyRDF.
```

```
df = P
```

```
df.Define(...).histoD(...).draw()
```

The configuration dictionary mixes PyRDF specific options with external backend options

```
conf_dict = {  
    # Provide Spark options  
    'spark.master': 'hostname:port',  
    'spark.executor.instances': 1,  
    # But also PyRDF options  
    'npartitions': 5,  
    'use_caching': True  
}
```



Connection step in Spark

The SparkContext is the entrypoint for the user to send computations to a Spark cluster:

```
from pyspark import SparkConf, SparkContext  
  
conf = {  
    'spark.master': 'hostname:port',  
    'spark.executor.instances': 1,  
}  
  
sparkConf = SparkConf().setAll(conf.items())  
  
sc = SparkContext(conf = sparkConf)  
  
sc.parallelize(range(100))  
    .map(map_function)  
    .reduce(reduce_function)
```



Connection step in Dask

In Dask there are at least two common ways of connecting to a cluster:

```
from dask.distributed import Client  
from dask import delayed  
  
def myfun(): ...  
  
if __name__ == "__main__":  
    client = Client("scheduler_address:port")  
    futures = client.map(myfun, range(100))  
    results = client.gather(futures)
```

```
from dask.distributed import Client, SSHCluster  
from dask import delayed  
  
def myfun(): ...  
  
if __name__ == "__main__":  
    cluster = SSHCluster(config_options)  
    client = Client(cluster)  
    futures = client.map(myfun, range(100))  
    results = client.gather(futures)
```



Connection step in Dask

In Dask there are at least two common ways of connecting to a cluster:

```
from dask.distributed import Client, SSHCluster  
from dask import delayed  
  
def myfun(): ...  
  
if __name__ == "__main__":  
    client = Client("scheduler_address:port")  
    futures = client.map(myfun, range(100))  
    results = client.gather(futures)
```

Give directly the Dask scheduler address, no extra config needed

```
from dask.distributed import Client, SSHCluster  
from dask import delayed  
  
def myfun(): ...  
  
if __name__ == "__main__":  
    cluster = SSHCluster(config_options)  
    client = Client(cluster)  
    futures = client.map(myfun, range(100))  
    results = client.gather(futures)
```



Connection step in Dask

In Dask there are at least two common ways of connecting to a cluster:

```
from dask.distributed import Client  
from dask import delayed  
  
def myfun(): ...  
  
if __name__ == "__main__":  
    client = Client("scheduler_address:port")  
    futures = client.map(myfun, range(100))  
    results = client.gather(futures)
```

Connect to an unmanaged cluster (SSH) or a cluster manager (K8s, YARN, HTCondor) explicitly, setting various options programmatically

```
if __name__ == "__main__":  
    cluster = SSHCluster(config_options)  
    client = Client(cluster)  
    futures = client.map(myfun, range(100))  
    results = client.gather(futures)
```



Coming back to PyRDF

Two main alternatives:

- ▶ Separate PyRDF config from backend config
- ▶ Configure backend first, then let PyRDF inherit it
- ▶ ... anything else comes to mind?

```
PyRDF.use("spark",
    # Backend opts in one dict
    { 'spark.master': 'hostname:port', ... },
    # PyRDF opts in another dict
    { 'npartitions': 100},
)
```

```
conf = { 'spark.master': 'hostname:port', ...}
sparkConf = SparkConf().setAll(conf.items())
sc = SparkContext(conf = sparkConf)
```

```
PyRDF.use(sc, pyrdf_conf_dict)
```



Coming back to PyRDF

Two main alternatives:

- ▶ Separate PyRDF config from backend config
- ▶ Configure backend first, then let PyRDF inherit it
- ▶ ... anything else comes to mind?

```
PyRDF.use("spark",  
          # Backend opts in one dict  
          {'spark.master': 'hostname:port',...},  
          # PyRDF opts in another dict  
          {'npartitions': 100},)
```

A slight variation:

```
conf = {'spark.master': 'hostname:port',...}  
sparkConf = SparkConf().setAll(conf.items())  
sc = SparkContext(conf = sparkConf)  
pyrdf_backend = PyRDF.backend.Spark(sc)  
PyRDF.use(pyrdf_backend, pyrdf_conf_dict)
```



How does it look for other backends?

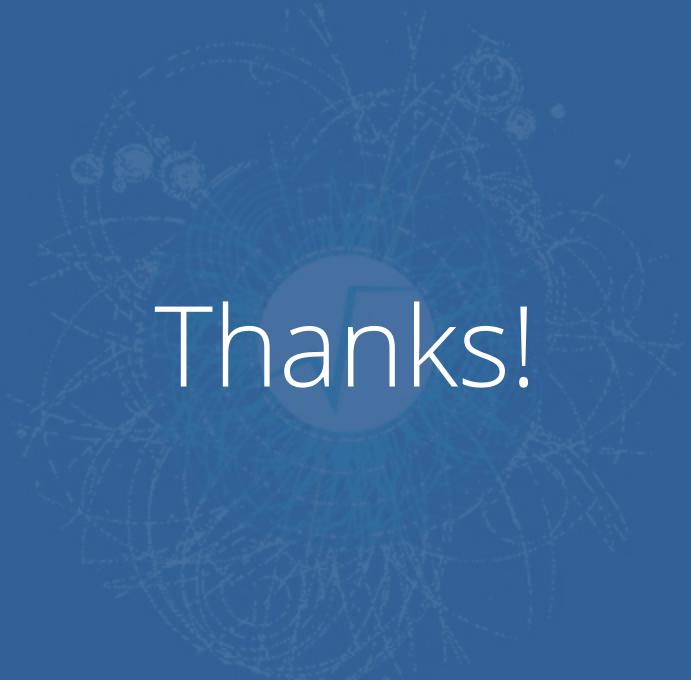
Spark

```
from pyspark import SparkConf, SparkContext  
  
import PyRDF  
  
conf = {'spark.master': 'hostname:port',...}  
sparkConf = SparkConf().setAll(conf.items())  
sc = SparkContext(conf = sparkConf)  
  
PyRDF.use(sc, pyrdf_conf_dict)  
  
pyrdf_backend = PyRDF.backend.Spark(sc)  
PyRDF.use(pyrdf_backend, pyrdf_conf_dict)
```

Dask

```
from dask.distributed import Client, SSHCluster  
  
import PyRDF  
  
if __name__ == "__main__":  
    cluster = SSHCluster(...)  
    client = Client(cluster)  
    dask.config.set({'foo.bar':'value'})  
    PyRDF.use(client, pyrdf_conf_dict)  
  
pyrdf_backend = PyRDF.backend.Dask(client)  
PyRDF.use(pyrdf_backend, pyrdf_conf_dict)
```

Note: "Local" backend would be the default, no need to call PyRDF.use



Thanks!