Spark

A Spark cheat sheet tailored to HOGWARTS can be found here.

The following sections describe some of the most common SPARK use cases.

Getting started - My first SPARK application

To get started with SPARK, create a Notebook in jupyhub and follow these steps.

1. Create a basic SPARK context

```
from pyspark import SparkContext
from pyspark.sql import SparkSession
import os

spark = SparkSession.builder.getOrCreate()
```

2. Create a SPARK dataframe from an array of data and display its schema

```
arrayData = [
        ('James',['Java','Scala'],{'hair':'black','eye':'brown'}),
        ('Michael',['Spark','Java',None],{'hair':'brown','eye':None}),
        ('Robert',['CSharp',''],{'hair':'red','eye':''}),
        ('Washington',None,None),
        ('Jefferson',['1','2'],{})
]

df = spark.createDataFrame(data=arrayData, schema = ['name','knownLanguages','properties'])
df.printSchema()
```

3. Show the dataframe using vertical layout and no truncation

```
df.show(vertical=True, truncate=False)
```

4. Write the dataframe on disk (in /tmp/people_dir) in parquet format

```
%rm -rf /tmp/people_dir
df.write.parquet('file:///tmp/people_dir')
%ls -lh /tmp/people_dir
```

5. Read the data from disk into a second dataframe and show it

```
df2 = spark.read.parquet('file:///tmp/people_dir')
df2.show()
```

6. Stop the SPARK context

```
spark.stop()
```

Congratulations, you have just completed your first SPARK application! Keep reading if you'd like to know how to access the datalake and/or the Hive data sources.

Using Spark UI in JupyterHub

To start a Spark master run this:

```
$SPARK_HOME/sbin/start-master.sh
```

This will start a Spark master in your jupyterhub, it runs on port 8080, if you want to use a different port, you can add the option `--webui-port=<other port>`. This web ui is accessible in jupyterhub at the url:

```
https://jupyhub.hogwarts.<environment>.azure.chimera.cyber.gc.ca/user/<last_name>_<first_name>_<initial>_/pro
```

You may now be noticing that there's no worker to run your Spark jobs. Run this command on jupyterhub:

```
$SPARK_HOME/sbin/start-slave.sh spark://$HOSTNAME:7077
```

To use this Spark instance in your jobs, set the Spark master URI to the resolved version of `spark://$HOSTNAME:7077`. You can do this in a few ways:

1.
   ```
   #!/bin/bash
   echo "spark://$HOSTNAME:7077"
   ```
   Copy the output of this and paste it into your Spark master URI
2. Or you can do it all in python:
   ```
   master_uri = f'spark://{os.environ.get("HOSTNAME")}:7077'
   ```

Accessing data sources stored in Hive

To access the data sources stored as Hive tables, create a Notebook in jupyhub and follow these steps.

1. Create a SPARK context with support for HIVE

```
from pyspark import SparkContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

spark = ( SparkSession.builder
            .enableHiveSupport()
            .getOrCreate()
        )
```

2. List all tables that start with 'geo'

```
spark.sql("show tables like 'geo*'").show(truncate=False)
```

3. Print the schema of the geo_quova table

```
spark.sql("select * from geo_quova").printSchema()
```

4. Perform a SELECT statement on that table

```
spark.sql("select start_ip_int, end_ip_int, country from geo_quova").show(10)
```

5. Register scala User Defined Functions (UDFs) and use them to convert IPs to Strings

```
from hogwarts.spark.sql.functions import ipv4_long_col_to_str, register
register(spark)

spark.sql("select ipv4_long_col_to_str(start_ip_int), ipv4_long_col_to_str(end_ip_int), country from geo_quov
```

6. Stop SPARK context

```
spark.stop()
```

Other useful pyspark sql methods

Rename columns using **as**:

```
spark.sql("select ipv4_long_col_to_str(start_ip_int) as start_ip, ipv4_long_col_to_str(end_ip_int) as end_ip,
```

Filter the results:

```
df = spark.sql("select ipv4_long_col_to_str(start_ip_int) as start_ip, ipv4_long_col_to_str(end_ip_int) as en
df.filter("country != 'united states'").show()
```

Convert a reasonable amount of rows (25 in the example below) to a Panda dataframe:

```
spark.sql("select ipv4_long_col_to_str(start_ip_int) as start_ip, ipv4_long_col_to_str(end_ip_int) as end_ip,
```

⚠ IMPORTANT: trying to convert a huge spark dataframe to a Panda dataframe will result in out of memory error.

Describe Hive table including detailed table information:

```
spark.sql("describe formatted geo_quova").show(100, truncate=False)
```

Reading from/Writing to the datalake

To access the datalake using SPARK, create a Notebook in jupyhub and follow these steps.

1. Create a basic SPARK context

```
from pyspark import SparkContext
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
```

2. Create a SPARK dataframe from an array of data and display its schema

```
arrayData = [
        ("23.10.80.250", "e9631.j.akamaiedge.net"),
        ("40.90.22.185", "fe-by01p-msa.com"),
        ("104.16.121.127", "medium.com"),
        ("69.16.175.10", "cds.r3t6j3w4.hwcdn.net"),
        ("13.226.139.75", "ui.playboyengineering.com"),
        ("31.13.71.49", "mmx-ds.cdn.whatsapp.net"),
        ("104.16.151.254", "www.vibe.com")
]

df = spark.createDataFrame(data=arrayData, schema = ['ip','hostname'])
df.printSchema()
```

3. Write a CSV to the **users** datalake container in the **demo/ip_to_hostname** folder

```
df.coalesce(1).write.mode("overwrite").csv("abfss://users@hogwartsdatalakeusersu.dfs.core.windows.net/demo/ip
```

4. Read back the CSV into a second dataframe called **df2** and show it

```
df2 = spark.read.format("csv").load("abfss://users@hogwartsdatalakeusersu.dfs.core.windows.net/demo/ip_to_hos
df2.printSchema()
```

```
df2.show(truncate=False)
```

5. Stop SPARK context

```
spark.stop()
```

The SPARK cluster

In the previous sections, we have used the mini-spark cluster bundled within jupyhub. This is very useful to develop and test your spark application locally but this instance is limited in number of cores and memory. So when you need to scale your analytic to handle more data, you can launch your application on the HOGWARTS SPARK cluster. That cluster is equipped with dozens of cores and hundreds of GiB of memory. To run your application on the cluster, simply add the SPARK master uri when creating your SPARK context:

```
spark = ( SparkSession.builder
            .master("spark://ver-1-spark-master-svc.spark:7077")
            .appName("my awesome app")
            .config("spark.executor.memory", "1g")
            .config("spark.cores.max", '2')
            .enableHiveSupport()
            .getOrCreate()
        )
```

Please note that you can specify the name of your SPARK application using `.appName`. This is the name that will appear in the SPARK console. The SPARK console can be accessed using a me2sa browser at: https://spark.u.hogwarts.azure.chi/ (https://spark.hogwarts.pb.azure.chimera.cyber.gc.ca/ on PB). You can look at the progress of your execution from that console.

⚠ IMPORTANT: when running your application on the SPARK cluster, you have to be careful about the number of resources that you request for your application (in order to share resources with your colleagues). You can limit the number of cores and memory that your application will use using the following config parameters: `"spark.executor.memory"` and `"spark.cores.max"`.

Using custom environment for Spark Workers

If you wish to use a custom environment for specific UDF, there is a possibility to send a packed environment to the workers before executing your functions.

As found in the Spark documentation, starting with Spark 3.1 you can add the following two lines to your code to specify which file to use.

```
os.environ['PYSPARK_PYTHON'] = "./mycustomenv/bin/python"

spark = ( SparkSession.builder
         .master("spark://ver-1-spark-master-svc.spark:7077")
         .appName("my awesome app")
         .config("spark.executor.memory", "1g")
         .config("spark.cores.max", '2')
         .config("spark.archives", f"abfss://users@{datalakename}.dfs.core.windows.net/path/to/custom_conda_
         .enableHiveSupport()
         .getOrCreate()
        )
```

The name “mycustomenv” can be modified, but needs to be the same between the PYSPARK_PYTHON value and the value specified after the '#' in the spark.archives configuration.

You can use libraries like dlbrowse to upload files to Azure Datalake.

A quick reminder on Conda environment:

```
#You can create an environment using
conda create -y -n custom_conda_env
#NOTE: hogwarts-pyspark is currently using python 3.8, so you may need to force conda to create an environme
conda create -y -n custom_conda_env python=3.8
#You can activate your environment using
conda activate custom_conda_env
#You can add libraries using
conda install requests==2.22.0
conda install conda-pack
#After having conda-pack installed (either in your env or elsewhere), you can package it
conda pack -f -o custom_conda_env.tar.gz
#You can get out of a conda environment using
conda deactivate
```

If you want to use a different version of python, you need to use the same version for both the Spark driver (your notebook, or airflow) and the Spark worker.

```
#Create an environment specifying a different version of python
conda create -y -n python36_conda_env_worker python=3.6 conda-pack
conda activate python36_conda_env_worker
conda install requests==2.23.0 #Could be done at create time
conda pack -f -o python36_conda_env_worker.tar.gz
conda deactivate

#Create a second environment for your driver, which will contain libraries that are not needed by the worker
conda create -y -n python36_conda_env_driver python=3.6
conda activate python36_conda_env_driver
#Install a version of pyspark that match the one deployed by DASI2B
conda install pyspark
conda install ipykernel
#Install your new conda environment as a kernel for JupyterHub
python -m ipykernel install --user --name=python36_conda_env_driver

#You can now check that the kernel was added using
jupyter kernelspec list

#Then you can change your Notebook kernel to use python36_conda_env_driver
```

Advanced topics

Create or replace a temporary view from a spark dataframe:

```
# Let's assume that we have a spark dataframe called df
df.createOrReplaceTempView('MY_TEMP_TABLE')
df2 = spark.sql('select count(*) from MY_TEMP_TABLE').show()
```

Create a User Defined Function (UDF):

```
df = spark.sql("select ip_long_to_str(start_ip_int) as start_ip, ip_long_to_str(end_ip_int) as end_ip, count

@udf("boolean")
def isInNorthAmerica(country):
    if(country == "united states" or country == "canada" or country == "mexico"):
        return True;
    else:
        return False;

df.withColumn("is_in_north_america", isInNorthAmerica(df.country)).show()
```

References

- PySpark SQL documentation
- PySpark SQL Cheat Sheet
- PySpark Style Guide
- HOGWARTS built-in User Defined Functions
    - Python examples that use these UDFs
- HOGWARTS Spark Cheat Sheet
- Hogwarts PySpark API