Task 3: Apache Spark & Kubernetes

Although for many aspects of this course, we focused on HPC clusters, these are not the only type of cluster that you may find in the wild. In the last decade, Big Data clusters have been deployed in both companies and academia to deal with the increasing amount of data. In this laboratory assignment, we will cover the state-of-the-art programming API for Big Data clusters: Apache Spark. We will focus on how to program and manipulate data, leaving aside how to deploy and manage a cluster of these characteristics or how to do performance tuning. In order to cover some basics of how to deploy it, we will rely on the orchestration software Kubernetes (k8s).

Part 1: Apache Spark

In this part, we will cover the basic elements of programming using a map-reduce methodology. For that purpose, we will be using Apache Spark as a reference, but bear in mind that similar frameworks exist, and principles can be extrapolated.

An overview of Apache Spark

- Spark is a **distributed computing platform** that operates on a cluster. We expect that nodes do NOT share a memory space, but they are connected using a high-speed dedicated network. Distributed filesystems such as HDFS, GlusterFS or Ceph, which work over the network, are extremely useful.
- It is considered the **next generation of previous map-reduce standard Apache Hadoop**. Main difference is the use of the memory instead of disk for intermediate operations, but there are many more improvements.
- **It is built on Java**. Despite this, it can be programmed using Java, Scala, Python or R. The complete API can only be found in JVM-based languages but the most frequent one is PySpark, since people is reluctant to use JVM-based languages in data science. Indeed, since Hadoop was only available for Java, it is likely that some Java codes of Spark are adaptations of previous Hadoop codes.
- **Resilient Distributed Dataset (RDD) is the basic unit** that is processed in Spark. Equivalent to a numpy array but distributed.
- **RDD API usually exposes the low-level operations of Apache Spark**, useful for preprocessing data but useless for data analytics
- **For data analysis, Dataframe API and Spark SQL** is used. It relies on a pandas-alike API that even accepts SQL code (which may sound crazy and useless for developers, but many *old* data scientists and statisticians are really proficient in SQL but not in Python).

Initializing Apache Spark

When using cloud Apache Spark environment or clusters with a Jupyter environments, it is likely that iPython kernels will already have some predefined variables: `spark`, the Spark session and `sc`, the Spark context. In case they are not created, we provide a small code snippet to quickly create them relying on local resources.

```
from pyspark.sql import SparkSession

APP_NAME = "CAP-lab3"
SPARK_URL = "local[*]"
spark = SparkSession.builder.appName(APP_NAME).master(SPARK_URL).ge
tOrCreate()
sc = spark.sparkContext
```

Generally speaking, spark is used to access Spark SQL (Dataframe) API whereas sc is the low-level API of RDDs. Before redeclaring them, it is advisable to check their existence since it is highly likely that at least the Spark Session already exists.

For our environment in Google Colab, we need to execute the previous code fragment.

Resilient Distributed Dataset API

The RDD API is simple enough that we only must deal with three types of functions:

1. Creation of RDDs: functions that with arguments from the Apache Spark Driver (the program we use to access the cluster) creates an RDD that may be spread along the cluster. These operations are usually lazy, but Spark may run some checks on the arguments. This initiates a Directed Acyclic Graph (DAG). Examples of functions are:

   ```
   array = sc.parallelize([1,2,3,4,5,6,7,8,9,10], num_parts)

   quijote = sc.textFile("elquijote.txt")
   ```

2. Transformations: functions that apply to an RDD and return an RDD. These operations are lazy. All transformations are added to the DAG to save the sequence of operations we want to be done. Some examples are:

   ```
   charsPerLine = quijote.map(lambda s: len(s))
   allWords = quijote.flatMap(lambda s: s.split())
   allWordsNoArticles = allWords.filter(lambda a: a.lower() not
   in ["el", "la"])
   allWordsUnique = allWords.map(lambda s: s.lower()).distinct()
   sampleWords = allWords.sample(withReplacement=True, fraction=
   0.2, seed=666)
   weirdSampling = sampleWords.union(allWordsNoArticles.sample(F
   alse, fraction=0.3))
   ```

3. Actions: functions that apply to an RDD and may return a value that is sent to the Driver. These operations trigger the execution of the whole DAG, from the parent node to the current action. Some of them are:

```
numLines = quijote.count()
numChars = charsPerLine.reduce(lambda a,b: a+b) # also charsP
erLine.sum()
sortedWordsByLength = allWordsNoArticles.takeOrdered(10, key=
lambda x: -len(x))
```

To obtain the DAG, one can call the method `toDebugString()`.

To end this section, it is necessary to analyze a special kind of RDD called Key-Value RDD (K-V RDD). These RDDs are composed of tuples where the first element of the tuple is called key and second value. Special functions such `join` or `reduceByKey` only apply to K-V RDDs.

Exercise

1. Read the material provided. Read the documentation of all functions used and classified them among the three aforementioned categories.
2. For each transformation, evaluate the size of the output RDD depending on the input RDD. Give particular code examples of the transformation achieving minimum/maximum sizes.
3. From the previous classification, highlight the ones that only apply to K-V RDDs. Provide a code example for all of them.
4. Describe the different forms of persistence of RDDs. How do you save an RDD to disk? Which formats are supported?

Spark SQL API

The RDD API is the foundation of Apache Spark high-level APIs. We will look to the Spark SQL API as an example of this. This API focuses on dealing with Dataframes, structured data where all registers have the same number of fields.

However, the same principles apply, and some SQL operations will not trigger execution of the underlying DAG. Some of the interesting transformations are: `df.withColumn, df.select, df.groupby, df.where, df.filter.`

Also, User-Defined Functions (UDFs) can be defined to parallelize an operation over a column.

```
from pyspark.sql.functions import udf # check this library for
already defined functions

def splitWords(e):
  return e.split(" ")

splitWords = udf(splitWords)
```

Furthermore, SQL can be directly used to perform the same operations. You only need to first declare the Dataframe as temporary view.

```
df.createOrReplaceTempView("reddit")
```

Once it has been declared, table named reddit can be used:

```
spark.sql("select * from reddit limit 10")
```

Lastly, actions must be used to retrieve the data from the cluster. This can be done by using `df.toPandas()` or `df.show()`.

Exercise

1. Read the material provided. Read the documentation of all functions.
2. For all the operations in the material provided, try to write the equivalent code using SQL.
3. Is the current implementation of the wordcloud using Spark to speed up the computations? Why?/Why not? Implement an improved version. Hint: `WordCloud(…).generate_from_frequencies`. Draw a wordcloud of your choice with data from Reddit.