Spark Preparation

We check if we are in Google Colab. If this is the case, install all necessary packages.

To run spark in Colab, we need to first install all the dependencies in Colab environment i.e. Apache Spark 3.3.2 with hadoop 3.2, Java 8 and Findspark to locate the spark in the system. The tools installation can be carried out inside the Jupyter Notebook of the Colab. Learn more from A Must-Read Guide on How to Work with PySpark on Google Colab for Data Scientists!

```
try:
  import google.colab
  IN_COLAB = True
except:
  IN_COLAB = False
```

```
if IN_COLAB:
    !apt-get install openjdk-8-jdk-headless -qq > /dev/null
    !wget -q https://dlcdn.apache.org/spark/spark-3.3.2/spark-3.3.2-bin-hadoop3.
    !tar xf spark-3.3.2-bin-hadoop3.tgz
    !mv spark-3.3.2-bin-hadoop3 spark
    !pip install -q findspark
    import os
    os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
    os.environ["SPARK_HOME"] = "/content/spark"
```

Start a Local Cluster

Use findspark.init() to start a local cluster. If you plan to use remote cluster, skip the findspark.init() and change the cluster_url according.

```
import findspark
findspark.init()
```

```
spark_url = 'local'
```

```
from pyspark.sql import SparkSession
from pyspark.sql import SQLContext

spark = SparkSession.builder\
        .master(spark_url)\
        .appName('Spark Tutorial')\
        .config('spark.ui.port', '4040')\
        .getOrCreate()
```

Spark Entry Points

```
sc = spark.sparkContext
```

```
spark
```

```
sc
```

Simple RDD Operations

There are 2 types of RDD operations, tranformation and action. Transformation is an operation applied on a RDD to create new RDD (or create a new RDD from data). Action is an operation applied on a RDD to perform computation and send the result back to driver.

Transformation Operations

- *sc.parallelize(data)* create an RDD from data
- *rdd.filter(func)* create a new rdd from existing rdd and keep only those elements that func is true

Action Operations

- *rdd.count()* count number of elements in an rdd
- *rdd.first()* get the frist element in the rdd
- *rdd.collect()* gather all elements in the rdd into a python list
- *rdd.take(n)* gather first n-th elements in the rdd into a python list

```
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)
```

```
rdd
```

```
n = rdd.count()
print('count = {0}'.format(n))
```

```
rdd.first()
```

```
l = rdd.collect()
print(l)
```

```
l = rdd.take(3)
print(l)
```

```
f_rdd = rdd.filter(lambda d: d > 2)
```

```
f_rdd.collect()
```

```
f_rdd.count()
```

RDD Operations - map and reduce

- *rdd.map(func)* -- **transformation** -- create a new rdd by performing function func on each element in an rdd
- *rdd.reduce(func)* -- **action** -- aggregate all elements in an rdd using function func

These two operations perform functions on rdd elements. The function can be provided using lambda function. We can supply any lambda function to map and reduce operations. For map operation, the function must take one input and return one output. For reduce operation, the function must take two inputs and return one output.

```
data = ['line 1', '2', 'more lines', 'last line']
```

```
lines = sc.parallelize(data)
```

```
print(lines)
```

```
print(lines.collect())
```

Count the length of each line in the RDD and store results in a new RDD

```
lineLengths = lines.map(lambda line: len(line))
print(lineLengths.collect())
```

Sum the lenght of lines in the RDD. As RDD is partitioned, this reduce operation performs in a parallel fashion.

```
totalLength = lineLengths.reduce(lambda a, b: a+b)
print(totalLength)
```

```
data = (1,2,3,4)
rdd = sc.parallelize(data)
rdd2 = rdd.map(lambda x: x*2)
print(rdd2.collect())
sum_val = rdd2.reduce(lambda a, b: a+b)
print('sum = {0}'.format(sum_val))
mul_val = rdd2.reduce(lambda a, b: a*b)
print('mul = {0}'.format(mul_val))
```

RDD Operations - aggregate

Aggregate is an action operation *rdd.aggregate(zeroValue, seqOp, combOp)* that:

- performs *seqOp* to *zeroValue* and all RDD elements -- this basically transforms all elements in RDD into the type of output value
- and then aggregates the transformed RDD elements using *combOp*

Note that reduce is a simple form of aggreate operation.

```
rdd.collect()
```

The following aggregate operation is basically a *rdd.reduce(lambda a, b: a+b)* as the type output value is an integer which is the same as the RDD elements

```
rdd.aggregate(0,
              lambda zero, e: zero+e,
              lambda a, b: a+b)
```

```
rdd.aggregate(0,
              lambda zero, e: zero+1,
              lambda a, b: a+b)
```

The following aggregate operation returns an order pairs of (x, y) where

- x is the sum of all elements in RDD
- y is the count of all elements in RDD

```
rdd.aggregate((0, 0),
              lambda zero, e: (zero[0]+e, zero[1]+1),
              lambda a, b: (a[0]+b[0], a[1]+b[1]))
```

```
lines.collect()
```

The following aggregate operation returns an order pairs of (x, y) where

- x is the concatenation of all elements in RDD
- y is the sum of the length of all elements in RDD

```
lines.aggregate(("", 0),
                lambda zero, e: (zero[0]+e, zero[1]+len(e)),
                lambda a, b: (a[0]+b[0], a[1]+b[1]))
```

```
lines.collect()
```

```
lines.reduce(lambda s1, s2: s1+s2)
```

Working with Text

Before running this example, make sure that a data file 'star-wars.txt' has been uploaded to content folder of this colab.

First, read the content of the file using sc.textFile(). This creates an rdd whose elements are lines in the input file.

```
sw = sc.textFile('star-wars.txt')
for line in sw.take(10):
    print('{0}: [{1}]'.format(len(line), line))
```

```
print('Total = {0} lines'.format(sw.count()))
```

Remove all blank lines and lower all characters in all lines.

```
nb_lines = sw.filter(lambda line: len(line) > 0)
print('Non blank line = {0} lines'.format(nb_lines.count()))
all_lowers = nb_lines.map(lambda line: line.lower())
for line in all_lowers.take(10):
    print('{0}: [{1}]'.format(len(line), line))
```

We can split each line into words. Note that if we use *map* each element in the output RDD from *map* is a list of words in each line. However, if we use *flatMap* lists in all lines are combined into an RDD of all words.

```
words_map = all_lowers.map(lambda line: line.split())
for l in words_map.take(5):
    print(l)
```

```
words = all_lowers.flatMap(lambda line: line.split())
for w in words.take(10):
    print(w)
```

To count the occurances of each word, we first transform a word into a pairwise (key, value) of (word, 1)

```
words.map(lambda word: (word, 1)).take(5)
```

After transformation, we can count the occurances using *reduceByKey* which perform reduce(function) for all elements with the same key

```
mappers = words.map(lambda word: (word, 1))
counts = mappers.reduceByKey(lambda x, y: x+y)
for wc in counts.take(10):
    print(wc)
```

```
spark.stop()
```