Intro to Big Data - Lab 2 (PySpark)

What we will cover in this lab:

- **Getting familiar with pyspark (Python API for Spark)**
- **Getting familiar with MapReduce**

Useful link:

http://spark.apache.org/docs/1.6.0/programming-guide.html

Your tasks:

- Launch pyspark (in command line, Hadoop shell):

```
pyspark
```

- Create and copy the file “ex1.txt” in Hadoop (in a directory named ”lab2”) containing the following lines:

  Today it is planned to learn Spark
  Spark will be taught today
  It is interesting
  Spark is powerful
  The fire throws sparks

- Print the number of lines containing the word “spark” :

```
>>> lines = sc.textFile(’lab2/ex1.txt’)

>>> spark_lines = lines.filter (

lambda x: x.lower().find(“spark”)!=-1)

>>> print spark_lines.count()

4
```

  Remark: `sc` in the above code is a variable that provides the SparkContext.

- Populate the “lab2” folder with several (at least 5) non-empty text files (possibly copied from the local linux machine);
- Create a spark Resilient Distributed Dataset (RDD) containing each line from the files in the HDFS “lab2” folder:

```
>>> lines = sc.textFile(’lab2’)
```

- Filter out the empty lines:

```
>>> lines_nonempty = lines.filter( lambda x: len(x) > 0 )
```

- At this point, no actual data is processed. Spark/PySpark evaluates lazily, so it's not until we extract result data from an RDD (or a chain of RDDs) that any actual processing will be done. The following code returns the number of non-empty lines:

```
>>> lines_nonempty.count()

8
```

- Find the 5 most commonly occurring words with their associated frequencies:

```
>>> words = lines_nonempty.flatMap(lambda x: x.split())

>>> wordcounts = words.map(lambda x: (x, 1)).reduceByKey(lambda x,y:x+y).ma
p(lambda x:(x[1],x[0])).sortByKey(False)

>>> wordcounts.take(5)

[(7,u’test’),(6,u’file’),(6,u’is’),(4,u’spark’),(2,u’inspiring’)]
```

To understand what's going on it's best to consider this program as a pipeline of transformations. Apart from the initial call to the textFile method of variable `sc` (SparkContext) to create the first resilient distributed dataset (RDD) by reading lines from each file in the specified directory on HDFS, subsequent calls transfrom each input RDD into a new output RDD. We'll consider a simple example where we start by creating an RDD with just two lines with `sc.parallelize`, rather than reading the data from files with `sc.textFile`, and trace what each step in our wordcount program does. The lines are a quote from a Dr Seuss story.

```
>>> lines = sc.parallelize(['Its fun to have fun,','but you have to know ho
w.'])

>>> wordcounts = lines.map( lambda x: x.replace(',',' ').replace('.',' ').r
eplace('-',' ').lower()) \

        .flatMap(lambda x: x.split()) \

        .map(lambda x: (x, 1)) \

        .reduceByKey(lambda x,y:x+y) \

        .map(lambda x:(x[1],x[0])) \

        .sortByKey(False)

>>> wordcounts.take(10)

[(2, 'to'), (2, 'fun'), (2, 'have'), (1, 'its'), (1, 'know'), (1, 'how'), (
1, 'you'), (1, 'but')]
```

1. **map (<function>)**

map returns a new RDD containing values created by applying the supplied function to each value in the original RDD

Here we use a lambda function which replaces some common punctuation characters with spaces and convert to lower case, producing a new RDD:

```
>>> r1 = lines.map( lambda x: x.replace(',',' ').replace('.',' ').replac
e('-',' ').lower())

>>> r1.take(10)

['its fun to have fun ', 'but you have to know how ']
```

2. **flatMap (<function>)**

flatMap applies a function which takes each input value and returns a list. Each value of the list becomes a new, separate value in the output RDD

In our example, the lines are split into words and then each word becomes a separate value in the output RDD:

```
>>> r2 = r1.flatMap(lambda x: x.split())

>>> r2.take(20)

['its', 'fun', 'to', 'have', 'fun', 'but', 'you', 'have', 'to', 'know', '
how']

>>>
```

3. **map (<function>)**

In this second map invocation, we use a function which replaces each original value in the input RDD with a 2-tuple containing the word in the first position and the integer value 1 in the second position:

```
>>> r3 = r2.map(lambda x: (x, 1))

>>> r3.take(20)

[('its', 1), ('fun', 1), ('to', 1), ('have', 1), ('fun', 1), ('but', 1),
('you', 1), ('have', 1), ('to', 1), ('know', 1), ('how', 1)]

>>>
```

4. **reduceByKey(<function>)**

Expect that the input RDD contains tuples of the form (<key>,<value>). Create a new RDD containing a tuple for each unique value of <key> in the input, where the value in the second position of the tuple is created by applying the supplied lambda function to the <value>s with the matching <key> in the input RDD

Here the key will be the word and lambda function will sum up the word counts for each word. The output RDD will consist of a single tuple for each unique word in the data, where the word is stored at the first position in the tuple and the word count is stored at the second position

```
>>> r4 = r3.reduceByKey(lambda x,y:x+y)

>>> r4.take(20)

[('fun', 2), ('to', 2), ('its', 1), ('know', 1), ('how', 1), ('you', 1),
('have', 2), ('but', 1)]
```

5. map (<function>)

map a lambda function to the data which will swap over the first and second values in each tuple, now the word count appears in the first position and the word in the second position

```
>>> r5 = r4.map(lambda x:(x[1],x[0]))

>>> r5.take(20)

[(2, 'fun'), (1, 'how'), (1, 'its'), (1, 'know'), (2, 'to'), (1, 'you'),
(1, 'but'), (2, 'have')]
```

6. sortByKey(ascending=True|False)

sort the input RDD by the key value (the value at the first position in each tuple)

In this example the first position stores the word count so this will sort the words so that the most frequently occurring words occur first in the RDD - the False parameter sets the sort order to descending.

```
>>> r6 = r5.sortByKey(ascending=False)

>>> r6.take(20)

[(2, 'fun'), (2, 'to'), (2, 'have'), (1, 'its'), (1, 'know'), (1, 'how')
, (1, 'you'), (1, 'but')]

>>>
```

Assignment:

1. Add up the sizes of all the lines in ”ex1.txt” using the map and reduce operations.

2. Use the reduceByKey() operation on key-value pairs to count how many times each line of text occurs in a file.

3. Use sortByKey() to sort the key-value pairs alphabetically.