Spark Project - 2

Transformations and Actions

In this project, we will dive deeper into how those mechanisms work, and explore a wider range of the functions built into the Spark core.

About the dataset:

The file hamlet.txt contains the entire text of Shakespeare's play Hamlet. Shakespeare is well-known for his unique writing style and arguably one of the most influential writers in history. Hamlet is one of his most popular plays.

1. The file is in pure text format, and not ready for analysis. Before we can proceed, we'll have to clean up and reformat the data.

We will use the textFile() method to read the data into an RDD raw_hamlet. To verify our method, we will display the first five elements of our RDD.

```
raw_hamlet = sc.textFile("hamlet.txt")
raw_hamlet.take(5)
```

The output that we will get is something like this:

```
Out[1]:
['hamlet@0\t\tHAMLET',
 'hamlet@8',
 'hamlet@9',
 'hamlet@10\t\tDRAMATIS PERSONAE',
 'hamlet@29']
```

2. We have noticed that the text file uses the tab character (\t) as a delimiter. We'll need to split the file on the tab delimiter and convert the results into an RDD that's more manageable. We will name the resulting RDD as split_hamlet.

```
split_hamlet = raw_hamlet.map(lambda x: x.split("\t"))
split_hamlet.take(10)
```

The output will show 10 lines of the resulting RDD:

```
[['hamlet@0', '', 'HAMLET'],
 ['hamlet@8'],
 ['hamlet@9'],
 ['hamlet@10', '', 'DRAMATIS PERSONAE'],
 ['hamlet@29'],
 ['hamlet@30'],
 ['hamlet@31', 'CLAUDIUS', 'king of Denmark. (KING CLAUDIUS:)'],
 ['hamlet@74'],
 ['hamlet@75', 'HAMLET', 'son to the late, and nephew to the present king.'],
 ['hamlet@131']]
```

Lambda functions are great for writing quick functions we can pass into PySpark methods with simple logic. They fall short when we need to write more customized logic, though. Fortunately, PySpark lets us define a function in Python first, then pass it in. Any function that returns a sequence of data in PySpark (versus a guaranteed Boolean value, like filter() requires) must use a yield statement to specify the values that should be pulled later.

Yield is a Python technique that allows the interpreter to generate data on the fly and pull it when necessary, instead of storing it to memory immediately. Because of its unique architecture, Spark takes advantage of this technique to reduce overhead and improve the speed of computations.

Spark runs the named function on every element in the RDD and restricts it in scope. Each instance of the function only has access to the object(s) you pass into the function, and the Python libraries available in your environment. If you try to refer to variables outside the scope of the function or import libraries, those actions may cause the computation to crash. That's because Spark compiles the function's code to Java to run on the RDD objects (which are also in Java). Not all functions require us to use yield; only the ones that generate a custom sequence of data do. For map() or filter(), we use return to return a value for every single element in the RDD we're running the functions on.

3. We'll use the flatMap() method with the named function hamlet_speaks to check whether a line in the play contains the text HAMLET in all caps (indicating that Hamlet spoke). flatMap() is different than map() because it doesn't require an output for every element in the RDD. The flatMap() method is useful whenever we want to generate a sequence of values from an RDD.

In this case, we want an RDD object that contains tuples of the unique line IDs and the text "hamlet speaketh!," but **only for the elements in the RDD that have "HAMLET" in one of the values**. We can't use the map() method for this because it requires a return value for every element in the RDD.

We want each element in the resulting RDD to have the following format:

1. The first value should be the unique line ID (e.g.'hamlet@0') , which is the first value in each of the elements in the split_hamlet RDD.
2. The second value should be the string "hamlet speaketh!"

```
def hamlet_speaks(line):
    id = line[0]
    speaketh = False

    if "HAMLET" in line:
        speaketh = True

    if speaketh:
        yield id,"hamlet speaketh!"

hamlet_spoken = split_hamlet.flatMap(lambda x: hamlet_speaks(x))
hamlet_spoken.take(10)
```

And the output will be:

```
Out[1]:
[('hamlet@0', 'hamlet speaketh!'),
 ('hamlet@75', 'hamlet speaketh!'),
 ('hamlet@1004', 'hamlet speaketh!'),
 ('hamlet@9144', 'hamlet speaketh!'),
 ('hamlet@12313', 'hamlet speaketh!'),
 ('hamlet@12434', 'hamlet speaketh!'),
 ('hamlet@12760', 'hamlet speaketh!'),
 ('hamlet@12858', 'hamlet speaketh!'),
 ('hamlet@14821', 'hamlet speaketh!'),
 ('hamlet@15261', 'hamlet speaketh!')]
```

4. The RDD hamlet_spoken now contains the line numbers for the lines where Hamlet spoke. While this is handy, we don't have the full line anymore. Instead, let's use a filter() with a named function to extract the original lines where Hamlet spoke. The functions we pass into filter() **must** return values, which will be either True or False.

```
def filter_hamlet_speaks(line):
    if "HAMLET" in line:
        return True
    return False

hamlet_spoken_lines = split_hamlet.filter(lambda line: filter_hamlet_speaks(line))
hamlet_spoken_lines.take(5)
```

With the output being:

```
[['hamlet@0', '', 'HAMLET'],
 ['hamlet@75', 'HAMLET', 'son to the late, and nephew to the present king.'],
 ['hamlet@1004', '', 'HAMLET'],
 ['hamlet@9144', '', 'HAMLET'],
 ['hamlet@12313',
  'HAMLET',
  '[Aside]  A little more than kin, and less than kind.']]
```

5. Spark has two kinds of methods, transformations and actions. While we've explored some of the transformations, we haven't used any actions other than take().

Whenever we use an action method, Spark forces the evaluation of lazy code. If we only chain together transformation methods and print the resulting RDD object, we'll see the type of RDD (e.g. a PythonRDD or PipelinedRDD object), but not the elements within it. That's because the computation hasn't actually happened yet.

Even though Spark simplifies chaining lots of transformations together, it's good practice to use actions to observe the intermediate RDD objects between those transformations. This will let you know whether your transformations are working the way you expect them to.

The count() method returns the number of elements in an RDD. count() is useful when we want to make sure the result of a transformation contains the right number of elements. To get the number of elements in the RDD hamlet_spoken_lines, we will run .count() on it:

Running .collect() on an RDD returns a list representation of it. To get a list of all the elements in hamlet_spoken_lines, we would write hamlet_spoken_lines.collect()

```
spoken_count = 0
spoken_101 = list()
spoken_count = hamlet_spoken_lines.count()
spoken_collect = hamlet_spoken_lines.collect()
spoken_101 = spoken_collect[100]
```

The output is stored in the spoken_count and spoken_collect variables respectively and can be viewed using the take() function anytime.

Till now, we dove deeper into transformation and action functions and used flatmap() function to serve a requirement different from map(). We also learnt the limitations of lambda functions for the purpose of more customized logic.

Part - II

In this next part of the project, we will convert the text of Hamlet into a format that is more suitable for data analysis.

1. The first value in each element (or line from the play) is a line number that identifies the line of the play the text is from. It appears in the following format:

'hamlet@0'
'hamlet@8',
'hamlet@9',
...

We don't need the hamlet@ at the beginning of these IDs for our data analysis. We will extract just the integer part of the ID from each line, which is much more useful. We will transform the RDD split_hamlet into a new RDD hamlet_with_ids which will contain the clean version of the line ID for each element.

```
raw_hamlet = sc.textFile("hamlet.txt")
split_hamlet = raw_hamlet.map(lambda line: line.split('\t'))
split_hamlet.take(5)

def clean(line):
    id = line[0].split('@')[1]
    results=[]
    results.append(id)
    if len(line) > 1:
        for y in line[1:]:
            results.append(y)
    return results

hamlet_with_ids = split_hamlet.map(lambda line: clean(line))
hamlet_with_ids.take(10)
```

The output will be:

```
[['0', '', 'HAMLET'],
 ['8'],
 ['9'],
 ['10', '', 'DRAMATIS PERSONAE'],
 ['29'],
 ['30'],
 ['31', 'CLAUDIUS', 'king of Denmark. (KING CLAUDIUS:)'],
 ['74'],
 ['75', 'HAMLET', 'son to the late, and nephew to the present king.'],
 ['131']]
```

2. Next, we will get rid of elements that don't contain any actual words (and just have an ID as the first value). These typically represent blank lines between paragraphs or sections in the play.

We also want to remove any blank values ('') within elements, which don't contain any useful information for our analysis.

```
hamlet_with_ids.take(5)
real_text = hamlet_with_ids.filter(lambda line: len(line) > 1)
hamlet_text_only = real_text.map(lambda line: [l for l in line if l != ''])
hamlet_text_only.take(10)
```

We can see the comparison of the old RDD with the new one in the output:

```
Out[1]: [['0', '', 'HAMLET'], ['8'], ['9'], ['10', '', 'DRAMATIS PERSONAE'], ['29']]
Out[1]:
[['0', 'HAMLET'],
 ['10', 'DRAMATIS PERSONAE'],
 ['31', 'CLAUDIUS', 'king of Denmark. (KING CLAUDIUS:)'],
 ['75', 'HAMLET', 'son to the late, and nephew to the present king.'],
 ['132', 'POLONIUS', 'lord chamberlain. (LORD POLONIUS:)'],
 ['177', 'HORATIO', 'friend to Hamlet.'],
 ['204', 'LAERTES', 'son to Polonius.'],
 ['230', 'LUCIANUS', 'nephew to the king.'],
 ['261', 'VOLTIMAND', '|'],
 ['273', '|']]
```

3. While previewing the RDD after each task, we noticed some pipe characters (|) in odd places that add no value for us. It may appear as a standalone value in an element, or as part of an otherwise useful string value.

So, we will remove list items of the RDD that contain only the pipe character and replace any pipe character that appears in strings with an empty character. The resulting RDD will be assigned to clean_hamlet.

```
hamlet_text_only.take(10)


def clean(line):
    results=[]
    for y in line:
        if y=="|":
            pass
        elif "|" in y:
            fmtd=y.replace("|","")
            results.append(fmtd)
        else:
            results.append(y)
    return results

clean_hamlet = hamlet_text_only.map(lambda x: clean(x))
clean_hamlet.take(10)
```

The output will show the first 10 lines of the clean RDD:

```
[['0', 'HAMLET'],
 ['10', 'DRAMATIS PERSONAE'],
 ['31', 'CLAUDIUS', 'king of Denmark. (KING CLAUDIUS:)'],
 ['75', 'HAMLET', 'son to the late, and nephew to the present king.'],
 ['132', 'POLONIUS', 'lord chamberlain. (LORD POLONIUS:)'],
 ['177', 'HORATIO', 'friend to Hamlet.'],
 ['204', 'LAERTES', 'son to Polonius.'],
 ['230', 'LUCIANUS', 'nephew to the king.'],
 ['261', 'VOLTIMAND'],
 ['273']]
```

This project gave us a good view of transformations and actions in spark and how they can be used to clean data and make it fit for Data Analysis.