5b_Transformations_others

February 21, 2022

0.1 Demonstrating transformations of RDDs

[1]:
```
# Import SparkContext and SparkConf
from pyspark import SparkContext, SparkConf
```

[2]:
```
# Initialize spark
conf = SparkConf().setAppName("LearnTransormations")
sc = SparkContext(conf=conf)
```

```
22/02/21 12:00:17 WARN Utils: Your hostname, ThinkCentre resolves to a loopback
address: 127.0.1.1; using 10.180.5.223 instead (on interface eno1)
22/02/21 12:00:17 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another
address
22/02/21 12:00:18 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform… using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
22/02/21 12:00:19 WARN Utils: Service 'SparkUI' could not bind on port 4040.
Attempting port 4041.
22/02/21 12:00:19 WARN Utils: Service 'SparkUI' could not bind on port 4041.
Attempting port 4042.
22/02/21 12:00:19 WARN Utils: Service 'SparkUI' could not bind on port 4042.
Attempting port 4043.
```

[3]:
```
# 4. Load a text file - A poem By Shri. Rabindranath Thagore
# My Dependence
rdd = sc.textFile("5b_mydependence.txt")
```

0.1.1 1. map & flatMap

[4]:
```
# Question1: Convert all words in a rdd to lowercase and
#            split the lines of a document using space.

# How to do it?

def convert(lines):
    lines = lines.lower() # Convert all the words to lower case
```

```
    lines = lines.split() # Split each line into words
    return lines

rdd1 = rdd.map(convert)

# List of list of words, each sublist belongs to a line
rdd1.collect()
```

[4]:
```
[['i', 'like', 'to', 'be', 'dependent,', 'and', 'so', 'for', 'ever'],
 ['with', 'warmth', 'and', 'care', 'of', 'my', 'mother'],
 ['my', 'father', ',', 'to', 'love,', 'kiss', 'and', 'embrace'],
 ['wear', 'life', 'happily', 'in', 'all', 'their', 'grace.'],
 [],
 ['i', 'like', 'to', 'be', 'dependent,', 'and', 'so', 'for', 'ever'],
 ['on', 'my', 'kith', 'and', 'kin,', 'for', 'they', 'all', 'shower'],
 ['harsh', 'and', 'warm', 'advices,', 'complaints'],
 ['full', 'wondering', ',true', 'and', 'info', 'giants.'],
 [],
 ['i', 'like', 'to', 'be', 'dependent,', 'and', 'so', 'for', 'ever'],
 ['for', 'my', 'friends,', 'chat', 'and', 'want', 'me', 'near'],
 ['with', 'domestic,family', 'and', 'romantic', 'tips'],
 ['colleagues', 'as', 'well', ',', 'guide', 'me', 'work', 'at', 'risks.'],
 [],
 ['i', 'like', 'to', 'be', 'dependent,', 'and', 'so', 'for', 'ever'],
 ['for', 'my', 'neighbours', 'too,', 'envy', 'at', 'times'],
 ['when', 'at', 'my', 'rise', 'of', 'fortune', 'like', 'to', 'hear'],
 ['my', 'daily', 'steps', ',', 'easy', 'and', 'odd', 'things', 'too.']]
```

[5]:
```
# Can we eliminate these sub lists? I need all words in a single list.
# How? Use flatmap.

def convert(lines):
    lines = lines.lower() # Convert all the words to lower case
    lines = lines.split() # Split each line into words
    return lines

rdd2 = rdd.flatMap(convert)

rdd2.collect() # A list of all words - flattened
```

[5]:
```
['i',
 'like',
 'to',
 'be',
 'dependent,',
```

```
'and',
'so',
'for',
'ever',
'with',
'warmth',
'and',
'care',
'of',
'my',
'mother',
'my',
'father',
',',
'to',
'love,',
'kiss',
'and',
'embrace',
'wear',
'life',
'happily',
'in',
'all',
'their',
'grace.',
'i',
'like',
'to',
'be',
'dependent,',
'and',
'so',
'for',
'ever',
'on',
'my',
'kith',
'and',
'kin,',
'for',
'they',
'all',
'shower',
'harsh',
'and',
'warm',
```

```
'advices,',
'complaints',
'full',
'wondering',
',true',
'and',
'info',
'giants.',
'i',
'like',
'to',
'be',
'dependent,',
'and',
'so',
'for',
'ever',
'for',
'my',
'friends,',
'chat',
'and',
'want',
'me',
'near',
'with',
'domestic,family',
'and',
'romantic',
'tips',
'colleagues',
'as',
'well',
',',
'guide',
'me',
'work',
'at',
'risks.',
'i',
'like',
'to',
'be',
'dependent,',
'and',
'so',
'for',
```

```
 'ever',
 'for',
 'my',
 'neighbours',
 'too,',
 'envy',
 'at',
 'times',
 'when',
 'at',
 'my',
 'rise',
 'of',
 'fortune',
 'like',
 'to',
 'hear',
 'my',
 'daily',
 'steps',
 ',',
 'easy',
 'and',
 'odd',
 'things',
 'too.']
```

0.1.2 2. take - What are the first 5 words?

[6]:
```
# Print the first 5 words and store in a list
L5 = rdd2.take(5)
L5
```

[6]:
```
['i', 'like', 'to', 'be', 'dependent,']
```

0.1.3 3. filter - Remove stop words

[7]:
```
# Remove stop words - a, an, the, am, are, ...

# Create a list of stopwords
stopwords = ['is','am','are','the','for','a','an']
rdd3 = rdd2.filter(lambda x: x not in stopwords)

rdd3.collect()
```

[7]:
```
['i',
 'like',
 'to',
 'be',
 'dependent,',
 'and',
 'so',
 'ever',
 'with',
 'warmth',
 'and',
 'care',
 'of',
 'my',
 'mother',
 'my',
 'father',
 ',',
 'to',
 'love,',
 'kiss',
 'and',
 'embrace',
 'wear',
 'life',
 'happily',
 'in',
 'all',
 'their',
 'grace.',
 'i',
 'like',
 'to',
 'be',
 'dependent,',
 'and',
 'so',
 'ever',
 'on',
 'my',
 'kith',
 'and',
 'kin,',
 'they',
 'all',
 'shower',
 'harsh',
```

```
'and',
'warm',
'advices,',
'complaints',
'full',
'wondering',
',true',
'and',
'info',
'giants.',
'i',
'like',
'to',
'be',
'dependent,',
'and',
'so',
'ever',
'my',
'friends,',
'chat',
'and',
'want',
'me',
'near',
'with',
'domestic,family',
'and',
'romantic',
'tips',
'colleagues',
'as',
'well',
',',
'guide',
'me',
'work',
'at',
'risks.',
'i',
'like',
'to',
'be',
'dependent,',
'and',
'so',
'ever',
```

```
 'my',
 'neighbours',
 'too,',
 'envy',
 'at',
 'times',
 'when',
 'at',
 'my',
 'rise',
 'of',
 'fortune',
 'like',
 'to',
 'hear',
 'my',
 'daily',
 'steps',
 ',',
 'easy',
 'and',
 'odd',
 'things',
 'too.']
```

0.1.4 4. groupby

The "groupBy" transformation will group the data in the original RDD. It creates a set of key value pairs: - key - output of a user function, - value - all items for which the function yields this key

We have to pass a function (in this case, I am using a lambda function) inside the "groupBy" which will take the first 3 characters of each word in "rdd3".

The key is the first 3 characters and value is all the words which start with these 3 characters.

0.1.5 Problem: Find out the words starting with a set of characters

[9]:
```
rdd4.collect()
```

[9]:
```
[('i', <pyspark.resultiterable.ResultIterable at 0x7fc3f13f9e80>),
 ('li', <pyspark.resultiterable.ResultIterable at 0x7fc3f1346280>),
 ('an', <pyspark.resultiterable.ResultIterable at 0x7fc3f1346ac0>),
 ('ev', <pyspark.resultiterable.ResultIterable at 0x7fc3f13463a0>),
 ('ca', <pyspark.resultiterable.ResultIterable at 0x7fc3f1346be0>),
 ('of', <pyspark.resultiterable.ResultIterable at 0x7fc3f1346400>),
```

```
('lo', <pyspark.resultiterable.ResultIterable at 0x7fc41c545eb0>),
('em', <pyspark.resultiterable.ResultIterable at 0x7fc3f134b4c0>),
('we', <pyspark.resultiterable.ResultIterable at 0x7fc3f134b430>),
('in', <pyspark.resultiterable.ResultIterable at 0x7fc3f134b550>),
('al', <pyspark.resultiterable.ResultIterable at 0x7fc3f134b5b0>),
('gr', <pyspark.resultiterable.ResultIterable at 0x7fc3f134b610>),
('co', <pyspark.resultiterable.ResultIterable at 0x7fc3f134b670>),
('gi', <pyspark.resultiterable.ResultIterable at 0x7fc3f134b6d0>),
('fr', <pyspark.resultiterable.ResultIterable at 0x7fc3f134b730>),
('ch', <pyspark.resultiterable.ResultIterable at 0x7fc3f134b790>),
('do', <pyspark.resultiterable.ResultIterable at 0x7fc3f134b7f0>),
('as', <pyspark.resultiterable.ResultIterable at 0x7fc3f134b850>),
('gu', <pyspark.resultiterable.ResultIterable at 0x7fc3f134b8b0>),
('at', <pyspark.resultiterable.ResultIterable at 0x7fc3f134b910>),
('ri', <pyspark.resultiterable.ResultIterable at 0x7fc3f134b970>),
('en', <pyspark.resultiterable.ResultIterable at 0x7fc3f134b9d0>),
('wh', <pyspark.resultiterable.ResultIterable at 0x7fc3f134ba30>),
('fo', <pyspark.resultiterable.ResultIterable at 0x7fc3f134ba90>),
('he', <pyspark.resultiterable.ResultIterable at 0x7fc3f134baf0>),
('da', <pyspark.resultiterable.ResultIterable at 0x7fc3f134bb50>),
('st', <pyspark.resultiterable.ResultIterable at 0x7fc3f134bbb0>),
('to', <pyspark.resultiterable.ResultIterable at 0x7fc3f134bc10>),
('be', <pyspark.resultiterable.ResultIterable at 0x7fc3f134bc70>),
('de', <pyspark.resultiterable.ResultIterable at 0x7fc3f134bcd0>),
('so', <pyspark.resultiterable.ResultIterable at 0x7fc3f134bd30>),
('wi', <pyspark.resultiterable.ResultIterable at 0x7fc3f134bd90>),
('wa', <pyspark.resultiterable.ResultIterable at 0x7fc3f134bdf0>),
('my', <pyspark.resultiterable.ResultIterable at 0x7fc3f134be50>),
('mo', <pyspark.resultiterable.ResultIterable at 0x7fc3f134beb0>),
('fa', <pyspark.resultiterable.ResultIterable at 0x7fc3f134bf10>),
(',', <pyspark.resultiterable.ResultIterable at 0x7fc3f134bf70>),
('ki', <pyspark.resultiterable.ResultIterable at 0x7fc3f134bfd0>),
('ha', <pyspark.resultiterable.ResultIterable at 0x7fc3f13e8a00>),
('th', <pyspark.resultiterable.ResultIterable at 0x7fc3f13e8f70>),
('on', <pyspark.resultiterable.ResultIterable at 0x7fc3f13e8be0>),
('sh', <pyspark.resultiterable.ResultIterable at 0x7fc3f13e8fa0>),
('ad', <pyspark.resultiterable.ResultIterable at 0x7fc3f13e8490>),
('fu', <pyspark.resultiterable.ResultIterable at 0x7fc3f13e8c10>),
('wo', <pyspark.resultiterable.ResultIterable at 0x7fc3f13e83a0>),
(',t', <pyspark.resultiterable.ResultIterable at 0x7fc3f1360040>),
('me', <pyspark.resultiterable.ResultIterable at 0x7fc3f13600a0>),
('ne', <pyspark.resultiterable.ResultIterable at 0x7fc3f1360100>),
('ro', <pyspark.resultiterable.ResultIterable at 0x7fc3f1360160>),
('ti', <pyspark.resultiterable.ResultIterable at 0x7fc3f13601c0>),
('ea', <pyspark.resultiterable.ResultIterable at 0x7fc3f1360220>),
('od', <pyspark.resultiterable.ResultIterable at 0x7fc3f1360280>)]
```

[8]:
```
# Find out the key, value pairs
# Example: First 2 chars is key and the
# words starting with these first two chars are value

# Example: ('wo', ['wondering', 'work'])

# If there is any word with only one or two characters, all those
# words will be returned as it is.

# Example: ('i', ['i', 'i', 'i', 'i'])
#          ('at', ['at', 'at', 'at'])

rdd4 = rdd3.groupBy(lambda w: w[0:2])
L = [(k, list(v)) for (k, v) in rdd4.collect()]
#print([(k, list(v)) for (k, v) in rdd4.take(1)])
print(L)
```

```
[('i', ['i', 'i', 'i', 'i']), ('li', ['like', 'life', 'like', 'like', 'like',
'like']), ('an', ['and', 'and', 'and', 'and', 'and', 'and', 'and', 'and', 'and',
'and', 'and', 'and']), ('ev', ['ever', 'ever', 'ever', 'ever']), ('ca',
['care']), ('of', ['of', 'of']), ('lo', ['love,']), ('em', ['embrace']), ('we',
['wear', 'well']), ('in', ['in', 'info']), ('al', ['all', 'all']), ('gr',
['grace.']), ('co', ['complaints', 'colleagues']), ('gi', ['giants.']), ('fr',
['friends,']), ('ch', ['chat']), ('do', ['domestic,family']), ('as', ['as']),
('gu', ['guide']), ('at', ['at', 'at', 'at']), ('ri', ['risks.', 'rise']),
('en', ['envy']), ('wh', ['when']), ('fo', ['fortune']), ('he', ['hear']),
('da', ['daily']), ('st', ['steps']), ('to', ['to', 'to', 'to', 'to', 'to',
'too,', 'to', 'too.']), ('be', ['be', 'be', 'be', 'be']), ('de', ['dependent,',
'dependent,', 'dependent,', 'dependent,']), ('so', ['so', 'so', 'so', 'so']),
('wi', ['with', 'with']), ('wa', ['warmth', 'warm', 'want']), ('my', ['my',
'my', 'my', 'my', 'my', 'my', 'my']), ('mo', ['mother']), ('fa', ['father']),
(',', [',', ',', ',', ',']), ('ki', ['kiss', 'kith', 'kin,']), ('ha', ['happily',
'harsh']), ('th', ['their', 'they', 'things']), ('on', ['on']), ('sh',
['shower']), ('ad', ['advices,']), ('fu', ['full']), ('wo', ['wondering',
'work']), (',t', [',true']), ('me', ['me', 'me']), ('ne', ['near',
'neighbours']), ('ro', ['romantic']), ('ti', ['tips', 'times']), ('ea',
['easy']), ('od', ['odd'])]
```

0.1.6 5. groupByKey, mapValues & sortByKey

a) groupByKey() "operates on Pair RDDs" and is used to group all the values related to a given key.

When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable) pairs.

- Hash-partitions the resulting RDD with the existing partitioner/parallelism level.

Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numPartitions argument to set a different number of tasks.

iterable: is an object that has an **iter** method which returns an iterator, or which defines a **getitem** method that can take sequential indexes starting from zero (and raises an IndexError when the indexes are no longer valid). So an iterable is an object that you can get an iterator from.

iterator: is an object with a next (Python 2) or _ *next* _ (Python 3) method.

Whenever you use a for loop, or map, or a list comprehension, etc. in Python, the next method is called automatically to get each item from the iterator, thus going through the process of iteration.

b) mapValues - For example, get the number of elements in a "value" of a "key" Signature: rdd3_grouped.mapValues(f)

Pass each value in the key-value pair RDD through a map function without changing the keys; this also retains the original RDD's partitioning.

x = sc.parallelize([("a", ["apple", "banana", "lemon"]), ("b", ["grapes"])])

def f(x): >return len(x)

x.mapValues(f).collect()

[('a', 3), ('b', 1)]

c) sortByKey: Signature:

sortByKey(ascending=True, numPartitions=None, keyfunc=<function RDD. at 0x7f0b28a192f0>)

Docstring:

Sorts this RDD, which is assumed to consist of (key, value) pairs.

Examples:

tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]

sc.parallelize(tmp).sortByKey().first()

('1', 3)

sc.parallelize(tmp).sortByKey(True, 1).collect()

[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]

sc.parallelize(tmp).sortByKey(True, 2).collect()

[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]

tmp2 = [(‘Mary’, 1), (‘had’, 2), (‘a’, 3), (‘little’, 4), (‘lamb’, 5)]

tmp2.extend([(‘whose’, 6), (‘fleece’, 7), (‘was’, 8), (‘white’, 9)])

sc.parallelize(tmp2).sortByKey(True, 3, keyfunc=lambda k:

k.lower()).collect()

[(‘a’, 3), (‘fleece’, 7), (‘had’, 2), (‘lamb’, 5),...(‘white’, 9), (‘whose’, 6)]

0.1.7 Question: How many times each word is present in the poem?

Hint: Use map, GroupByKey, mapValues

[10]:
```
# Question: How many times each word is present in the poem?

# first converting "rdd3" into "rdd3_mapped".
# The "rdd3_mapped" is nothing but a mapped (key,val) pair RDD.
# If a word "like" is present then it returns ('like', 1)

rdd3_mapped = rdd3.map(lambda x: (x,1))
rdd3_mapped.take(20)
```

[10]:
```
[('i', 1),
 ('like', 1),
 ('to', 1),
 ('be', 1),
 ('dependent,', 1),
 ('and', 1),
 ('so', 1),
 ('ever', 1),
 ('with', 1),
 ('warmth', 1),
 ('and', 1),
 ('care', 1),
 ('of', 1),
 ('my', 1),
 ('mother', 1),
 ('my', 1),
 ('father', 1),
 (',', 1),
 ('to', 1),
 ('love,', 1)]
```

[11]:
```
# groupByKey works on pairRDDs - RDD with (key, value) pairs
# applying "groupByKey" transformation on "rdd3_mapped" to group
# the all elements based on the keys - the keys are words
rdd3_grouped = rdd3_mapped.groupByKey()
rdd3_grouped.collect()
```

```
print(list((j[0], list(j[1])) for j in rdd3_grouped.take(10)))
```

```
[('i', [1, 1, 1, 1]), ('like', [1, 1, 1, 1, 1]), ('dependent,', [1, 1, 1, 1]),
('ever', [1, 1, 1, 1]), ('of', [1, 1]), ('father', [1]), ('wear', [1]),
('happily', [1]), ('in', [1]), ('kith', [1])]
```

[12]:
```
#
rdd3_freq = rdd3_grouped.mapValues(sum).sortByKey()
rdd3_freq.take(10)
```

[12]:
```
[(',', 3),
 (',true', 1),
 ('advices,', 1),
 ('all', 2),
 ('and', 12),
 ('as', 1),
 ('at', 3),
 ('be', 4),
 ('care', 1),
 ('chat', 1)]
```

0.1.8 6. reduceByKey - Use this to solve the above problem

Signature: reduceByKey(func, numPartitions=None, partitionFunc=<function portable_hash at 0x7f0b403def28>)

Docstring: Merge the values for each key using an associative and commutative reduce function. Reduce function is mandatary.

- This will also perform the merging locally on each mapper before sending results to a reducer.
- Output will be partitioned with C{numPartitions} partitions, or the default parallelism level

  from operator import add

  rdd = sc.parallelize([(“a”, 1), (“b”, 1), (“a”, 1)])

  sorted(rdd.reduceByKey(add).collect())

  [(‘a’, 2), (‘b’, 1)]

[15]:
```
# first converting "rdd3" into "rdd3_mapped".
# The "rdd3_mapped" is nothing but a mapped (key,val) pair RDD.
rdd3_mapped = rdd3.map(lambda x: (x,1))
rdd3_mapped.take(5)
```

[15]:
```
[('i', 1), ('like', 1), ('to', 1), ('be', 1), ('dependent,', 1)]
```

[16]:
```
from operator import add
reduceRDD = rdd3_mapped.reduceByKey(add)
```

```
reduceRDD.collect()
```

[16]:
```
[('i', 4),
 ('like', 5),
 ('dependent,', 4),
 ('ever', 4),
 ('of', 2),
 ('father', 1),
 ('wear', 1),
 ('happily', 1),
 ('in', 1),
 ('kith', 1),
 ('advices,', 1),
 ('full', 1),
 ('chat', 1),
 ('near', 1),
 ('romantic', 1),
 ('tips', 1),
 ('as', 1),
 ('guide', 1),
 ('work', 1),
 ('at', 3),
 ('risks.', 1),
 ('neighbours', 1),
 ('envy', 1),
 ('when', 1),
 ('hear', 1),
 ('steps', 1),
 ('things', 1),
 ('too.', 1),
 ('to', 6),
 ('be', 4),
 ('and', 12),
 ('so', 4),
 ('with', 2),
 ('warmth', 1),
 ('care', 1),
 ('my', 7),
 ('mother', 1),
 (',', 3),
 ('love,', 1),
 ('kiss', 1),
 ('embrace', 1),
 ('life', 1),
 ('all', 2),
 ('their', 1),
 ('grace.', 1),
```

```
 ('on', 1),
 ('kin,', 1),
 ('they', 1),
 ('shower', 1),
 ('harsh', 1),
 ('warm', 1),
 ('complaints', 1),
 ('wondering', 1),
 (',true', 1),
 ('info', 1),
 ('giants.', 1),
 ('friends,', 1),
 ('want', 1),
 ('me', 2),
 ('domestic,family', 1),
 ('colleagues', 1),
 ('well', 1),
 ('too,', 1),
 ('times', 1),
 ('rise', 1),
 ('fortune', 1),
 ('daily', 1),
 ('easy', 1),
 ('odd', 1)]
```

0.1.9 7. sortByKey

Problem: Print the list of words in descending order of their frequencies

[17]:
```
# Reverse (Key, value) to (value, Key)
# Because we want to sort on the frequency, not on word
# So, frequency should become key now, word will be value
revRDD = reduceRDD.map(lambda x:(x[1],x[0]))
#revRDD.collect()
```

[18]:
```
# Now, apply sortByKey
revRDD.sortByKey(False).take(10)
```

[18]:
```
[(12, 'and'),
 (7, 'my'),
 (6, 'to'),
 (5, 'like'),
 (4, 'i'),
 (4, 'dependent,'),
 (4, 'ever'),
 (4, 'be'),
 (4, 'so'),
 (3, 'at')]
```

0.1.10 Difference between groupByKey & reduceByKey

reduceByKey:

1. combine output with a common key on each partition before shuffling the data. In the diagram, there are three partitions, pairs on the same machine with the same key are combined (by using the lamdba function passed into reduceByKey) before the data is shuffled.

2. Then the lamdba function is called again to reduce all the values from each partition to produce one final result.

groupByByKey: (More shuffle operations)

1. All the key-value pairs are shuffled around. This is a lot of unnessary data to being transferred over the network.

2. To determine which machine to shuffle a pair to, Spark calls a partitioning function on the key of the pair. Spark spills data to disk when there is more data shuffled onto a single executor machine than can fit in memory.

[ ]: