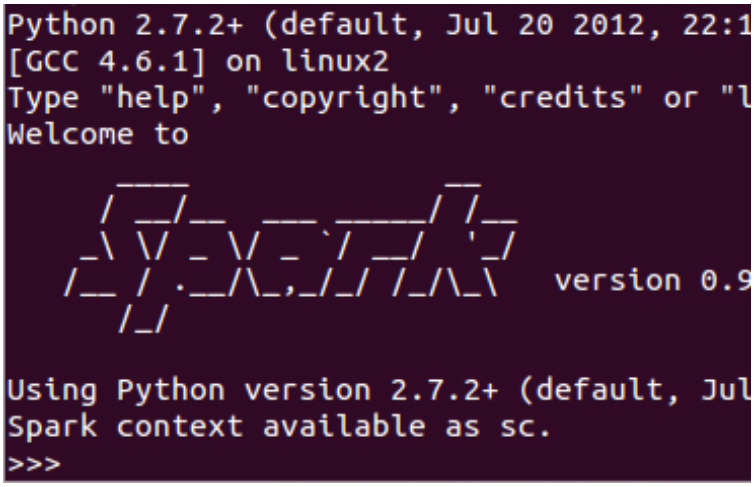
`

Getting started with PySpark - Part 2



In Part 1 (./../../blog/pyspark/index.html) we looked at installing the data processing engine Apache Spark (https://spark.incubator.apache.org/) and started to explore some features of its Python API, PySpark (http://spark.apache.org/docs/0.9.0/python-programming-guide.html). In this article, we look in more detail at using PySpark.

Revisiting the wordcount example

Recall the example described in Part 1 (../pyspark/index.html), which performs a wordcount on the documents stored under folder /user/dev/gutenberg on HDFS. We start by writing the transformation in a single invocation, with a few changes to deal with some punctuation characters and convert the text to lower case.

```
>>> wordcounts = sc.textFile('hdfs://ubuntu1:54310/user/dev/gutenberg') \
        .map( lambda x: x.replace(',',' ').replace('.',' ').replace('-',' ').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)
[(500662, u'the'), (331864, u'and'), (289323, u'of'), (196741, u'to'),
 (149380, u'a'), (132609, u'in'), (100711, u'that'), (92052, u'i'),
 (77469, u'he'), (72301, u'for')]
```

To understand whats going on its 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 (http://en.wikipedia.org/wiki/Dr._Seuss) story.

```
>>> lines = sc.parallelize(['Its fun to have fun,','but you have to know how.'])
>>> wordcounts = lines.map( lambda x: x.replace(',',' ').replace('.',' ').replace('-',' '
).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('.',' ').replace('-',' ').l
   ower())
   >>> 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 (pass

```
>>> r6 = r5.sortByKey(ascending=False)
>>> r6.take(20)
[(2, 'fun'), (2, 'to'), (2, 'have'), (1, 'its'), (1, 'know'), (1, 'how'), (1, 'you')
, (1, 'but')]
>>>
```

Finding frequent word bigrams

For a slightly more complicated task, lets look into splitting up sentences from our documents into word bigrams. A bigram (http://en.wikipedia.org/wiki/Bigram) is pair of successive tokens in some sequence. We will look at building bigrams from the sequences of words in each sentence, and then try to find the most frequently occuring ones.

The first problem is that values in each partition of our initial RDD describe lines from the file rather than sentences. Sentences may be split over multiple lines. The glom() RDD method is used to create a single entry for each document containing the list of all lines, we can then join the lines up, then resplit them into sentences using "." as the separator, using flatMap so that every object in our RDD is now a sentence.

```
>>> sentences = sc.textFile('hdfs://ubuntu1:54310/user/dev/gutenberg') \
    .glom() \
    .map(lambda x: " ".join(x)) \
    .flatMap(lambda x: x.split("."))
```

Now we have isolated each sentence we can split it into a list of words and extract the word bigrams from it. Our new RDD contains tuples containing the word bigram (itself a tuple containing the first and second word) as the first value and the number 1 as the second value.

```
>>> bigrams = sentences.map(lambda x:x.split()) \
    .flatMap(lambda x: [((x[i],x[i+1]),1) for i in range(0,len(x)-1)])
```

Finally we can apply the same reduceByKey and sort steps that we used in the wordcount example, to count up the bigrams and sort them in order of descending frequency. In reduceByKey the key is not an individual word but a bigram.

```
>>> freq_bigrams = bigrams.reduceByKey(lambda x,y:x+y) \
    .map(lambda x:(x[1],x[0])) \
    .sortByKey(False)
>>> freq_bigrams.take(10)
[(73228, (u'of', u'the')), (36008, (u'in', u'the')), (23860, (u'to', u'the')),
 (20582, (u'and', u'the')), (11534, (u'to', u'be')), (10944, (u'on', u'the')),
 (10548, (u'for', u'the')), (10374, (u'0', u'0')), (10117, (u'from', u'the')),
 (9983, (u'with', u'the'))]
```

RDD partitions, map and reduce

In the example above, the map and reduceByKey RDD transformations will be immediately recognizable to aficionados of the MapReduce paradigm. Spark supports the efficient parallel application of map and reduce operations by dividing data up into multiple partitions. In the example above, each file will by default generate one partition. What Spark adds to existing frameworks like Hadoop are the ability to add multiple map and reduce tasks to a single workflow.

There are some useful ways to look at the distribution of objects in each partition in our rdd:

```
>>> lines = sc.textFile('hdfs://ubuntu1:54310/user/dev/gutenberg')
>>> def countPartitions(id,iterator):
        c = 0
        for _ in iterator:
             c += 1
        yield (id,c)
>>> lines.mapPartitionsWithSplit(countPartitions).collectAsMap()
{0: 566, 1: 100222, 2: 124796, 3: 3735, ..., 96: 6690, 97: 3921, 98: 16271, 99: 1138}
>>>
```

- Each partition within an RDD is replicated across multiple workers running on different nodes in a cluster so that failure of a single worker should not cause the RDD to become unavailable.
- Many operations including map and flatMap operations can be applied independently to each partition, running as concurrent jobs based on the number of available cores. Typically these operations will preserve the number of partitions.
- When processing reduceByKey, Spark will create a number of output partitions based on the default paralellism based on the numbers of nodes and cores available to Spark. Data is effectively reshuffled so that input data from different input partitions with the same key value is passed to the same output partition and combined there using the specified reduce function. sortByKey is another operation which transforms N input partitions to M output partitions.

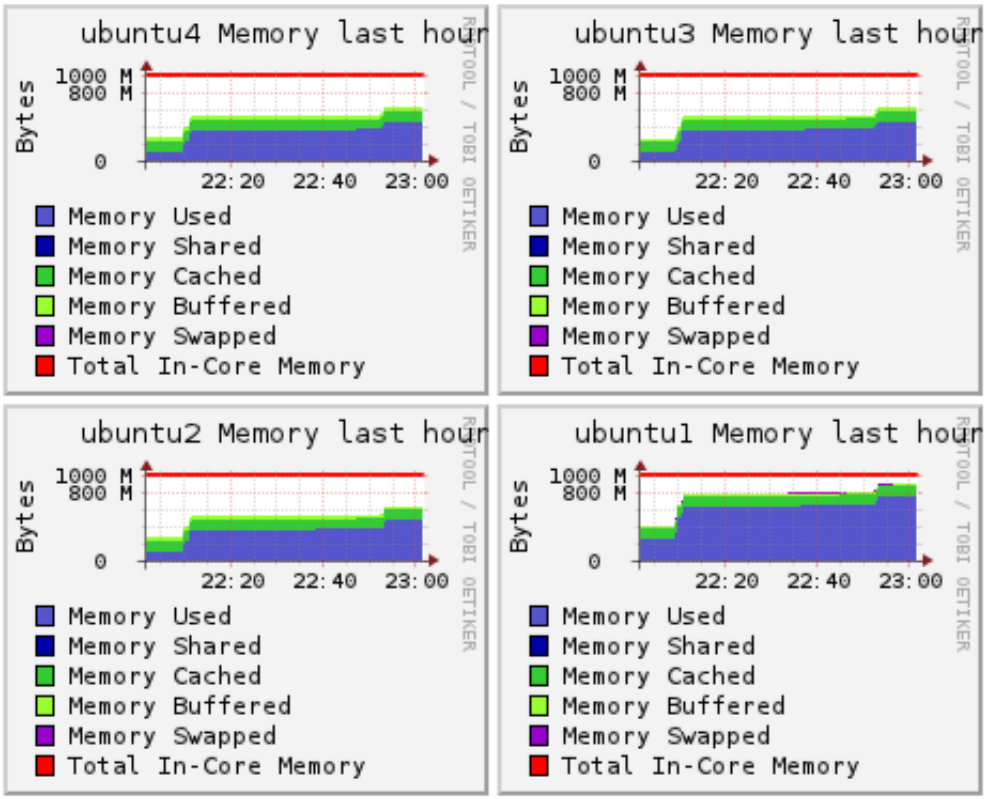
```
>>> sc.defaultParallelism
4
>>> wordcounts = sc.textFile('hdfs://ubuntu1:54310/user/dev/gutenberg') \
            .map( lambda x: x.replace(',',' ').replace('.',' ').replace('-',' ').lower())
 \
            .flatMap(lambda x: x.split()) \
            .map(lambda x: (x, 1)) \
            .reduceByKey(lambda x,y:x+y)
>>> wordcounts.mapPartitionsWithSplit(countPartitions).collectAsMap()
{0: 122478, 1: 122549, 2: 121597, 3: 122587}
```

The number of partitions generated by the reduce stage can be controlled by supplying the desired number of partitions as an extra parameter to reduceByKey:

```
>>> wordcounts = sc.textFile('hdfs://ubuntu1:54310/user/dev/gutenberg') \
             .map( lambda x: x.replace(',',' ').replace('.',' ').replace('-',' ').lower()
) \
             .flatMap(lambda x: x.split()) \
             .map(lambda x: (x, 1)) \
             .reduceByKey(lambda x,y:x+y,numPartitions=2)
>>> wordcounts.mapPartitionsWithSplit(countPartitions).collectAsMap()
{0: 244075, 1: 245136}
```

Monitoring memory usage

Spark/PySpark work best when there is sufficient resources to keep all the data in RDDs loaded in physical memory. In practice I found its best to carefully monitor whats happening with memory on each machine in the cluster. Although Spark's web pages offer a lot of information on task progress, I've found that installing and running Ganglia (http://ganglia.sourceforge.net/) provides a great way to monitor memory across the network.



Further reading

For more information on PySpark I suggest taking a look at some of these links:

- PySpark: Python API for Spark (YouTube, presentation by Josh Rosen) (http://www.youtube.com/watch?v=xc7Lc8RA8wE).
- Spark Python Programming Guide (http://spark.apache.org/docs/0.9.0/python-programming-guide.html)
- Resilient Distributed Dataset (RDD) API documentation (http://spark.incubator.apache.org/docs/latest/api/pyspark/pyspark.rdd.RDD-class.html)

7 Comments

Leave a comment

Anti-Spam Check

Hexadecimal number

0xd

Decimal equivalent

Comment

Name

Comment

Submit