4 Spark Basics 2

October 11, 2018

1 Spark Basics 2

1.1 Chaining

We can **chain** transformations and aaction to create a computation **pipeline**
Suppose we want to compute the sum of the squares

$$\sum_{i=1}^{n} x_i^2$$

where the elements x_i are stored in an RDD.

```
In [1]: #start the SparkContext
        import findspark
        findspark.init()

        from pyspark import SparkContext
        sc = SparkContext(master="local[4]")
        print(sc)
```

```
<SparkContext master=local[4] appName=pyspark-shell>
```

1.1.1 Create an RDD

```
In [2]: B=sc.parallelize(range(4))
        B.collect()
```

```
Out[2]: [0, 1, 2, 3]
```

1.1.2 Sequential syntax for chaining

Perform assignment after each computation

```
In [3]: Squares=B.map(lambda x:x*x)
        Squares.reduce(lambda x,y:x+y)
```

```
Out[3]: 14
```

1.1.3 Cascaded syntax for chaining

Combine computations into a single cascaded command

```
In [4]: B.map(lambda x:x*x)\
             .reduce(lambda x,y:x+y)
```

```
Out[4]: 14
```

1.1.4 Both syntaxes mean exactly the same thing

The only difference: * In the sequential syntax the intermediate RDD has a name `Squares` * In the cascaded syntax the intermediate RDD is *anonymous*

The execution is identical!

1.1.5 Sequential execution

The standard way that the map and reduce are executed is * perform the map * store the resulting RDD in memory * perform the reduce

1.1.6 Disadvantages of Sequential execution

1. Intermediate result (`Squares`) requires memory space.
2. Two scans of memory (of `B`, then of `Squares`) - double the cache-misses.

1.1.7 Pipelined execution

Perform the whole computation in a single pass. For each element of `B` 1. Compute the square 2. Enter the square as input to the `reduce` operation.

1.1.8 Advantages of Pipelined execution

1. Less memory required - intermediate result is not stored.
2. Faster - only one pass through the Input RDD.

1.1.9 Lazy Evaluation

This type of pipelined evaluation is related to **Lazy Evaluation**. The word **Lazy** is used because the first command (computing the square) is not executed immediately. Instead, the execution is delayed as long as possible so that several commands are executed in a single pass.

The delayed commands are organized in an **Execution plan**

For more on Pipelined execution, Lazy evaluation and Execution Plans see spark programming guide/RDD operations

1.1.10 An instructive mistake

Here is another way to compute the sum of the squares using a single reduce command. Can you figure out how it comes up with this unexpected result?

```
In [5]: C=sc.parallelize([1,1,2])
        C.reduce(lambda x,y: x*x+y*y)
```

```
Out[5]: 8
```

Answer:

1. `reduce` first operates on the pair $(1,1)$, replacing it with $1^2 + 1^2 = 2$
2. `reduce` then operates on the pair $(2,2)$, giving the final result $2^2 + 2^2 = 8$

1.2 getting information about an RDD

RDD's typically have hundreds of thousands of elements. It usually makes no sense to print out the content of a whole RDD. Here are some ways to get manageable amounts of information about an RDD

Create an RDD of length `n` which is a repetition of the pattern `1,2,3,4`

```
In [6]: n=1000000
        B=sc.parallelize([1,2,3,4]*int(n/4))
```

```
In [7]: #find the number of elements in the RDD
        B.count()
```

```
Out[7]: 1000000
```

```
In [8]: # get the first few elements of an RDD
        print('first element=',B.first())
        print('first 5 elements = ',B.take(5))
```

```
first element= 1
first 5 elements =  [1, 2, 3, 4, 1]
```

1.2.1 Sampling an RDD

- RDDs are often very large.
- Aggregates, such as averages, can be approximated efficiently by using a sample.
- Sampling is done in parallel and requires limited computation.

The method `RDD.sample(withReplacement,p)` generates a sample of the elements of the RDD. where - `withReplacement` is a boolean flag indicating whether or not a an element in the RDD can be sampled more than once. - `p` is the probability of accepting each element into the sample. Note that as the sampling is performed independently in each partition, the number of elements in the sample changes from sample to sample.

```
In [29]: # get a sample whose expected size is m
         # Note that the size of the sample is different in different runs
         m=5.
         print('sample1=',B.sample(False,m/n).collect())
         print('sample2=',B.sample(False,m/n).collect())
```

```
sample1= [4, 4, 4]
sample2= [2, 2]
```

1.2.2 Things to note and think about

- Each time you run the previous cell, you get a different estimate
- The accuracy of the estimate is determined by the size of the sample $n * p$
- See how the error changes as you vary p
- Can you give a formula that relates the variance of the estimate to $(p * n)$? (The answer is in the Probability and statistics course).

1.2.3 filtering an RDD

The method `RDD.filter(func)` Return a new dataset formed by selecting those elements of the source on which func returns true.

```
In [10]: print('the number of elements in B that are > 3 =',B.filter(lambda n: n > 3).count())
```

```
the number of elements in B that are > 3 = 250000
```

1.2.4 Removing duplicate elements from an RDD

The method `RDD.distinct()` Returns a new dataset that contains the distinct elements of the source dataset.

This operation requires a **shuffle** in order to detect duplication across partitions.

```
In [11]: # Remove duplicate element in DuplicateRDD, we get distinct RDD
         DuplicateRDD = sc.parallelize([1,1,2,2,3,3])
         print('DuplicateRDD=',DuplicateRDD.collect())
         print('DistinctRDD = ',DuplicateRDD.distinct().collect())
```

```
DuplicateRDD= [1, 1, 2, 2, 3, 3]
DistinctRDD =  [1, 2, 3]
```

1.2.5 flatmap an RDD

The method `RDD.flatMap(func)` is similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).

```
In [12]: text=["you are my sunshine","my only sunshine"]
         text_file = sc.parallelize(text)
         # map each line in text to a list of words
         print('map:',text_file.map(lambda line: line.split(" ")).collect())
         # create a single list of words by combining the words from all of the lines
         print('flatmap:',text_file.flatMap(lambda line: line.split(" ")).collect())
```

```
map: [['you', 'are', 'my', 'sunshine'], ['my', 'only', 'sunshine']]
flatmap: ['you', 'are', 'my', 'sunshine', 'my', 'only', 'sunshine']
```

1.2.6 Set operations

In this part, we explore set operations including **union**,**intersection**,**subtract**, **cartesian** in pyspark

```
In [13]: rdd1 = sc.parallelize([1, 1, 2, 3])
         rdd2 = sc.parallelize([1, 3, 4, 5])
```

1. union(other)

- Return the union of this RDD and another one.
- Note that that repetitions are allowed. The RDDs are **bags** not **sets**
- To make the result a set, use `.distinct`

```
In [14]: rdd2=sc.parallelize(['a','b',1])
         print('rdd1=',rdd1.collect())
         print('rdd2=',rdd2.collect())
         print('union as bags =',rdd1.union(rdd2).collect())
         print('union as sets =',rdd1.union(rdd2).distinct().collect())
```

```
rdd1= [1, 1, 2, 3]
rdd2= ['a', 'b', 1]
union as bags = [1, 1, 2, 3, 'a', 'b', 1]
union as sets = [1, 'a', 2, 3, 'b']
```

2. intersection(other)

- Return the intersection of this RDD and another one. The output will not contain any duplicate elements, even if the input RDDs did.Note that this method performs a shuffle internally.

```
In [15]: rdd2=sc.parallelize([1,1,2,5])
         print('rdd1=',rdd1.collect())
         print('rdd2=',rdd2.collect())
         print('intersection=',rdd1.intersection(rdd2).collect())
```

```
rdd1= [1, 1, 2, 3]
rdd2= [1, 1, 2, 5]
intersection= [1, 2]
```

3. subtract(other, numPartitions=None)

- Return each value in self that is not contained in other.

```
In [16]: print('rdd1=',rdd1.collect())
         print('rdd2=',rdd2.collect())
         print('rdd1.subtract(rdd2)=',rdd1.subtract(rdd2).collect())
```

```
rdd1= [1, 1, 2, 3]
rdd2= [1, 1, 2, 5]
rdd1.subtract(rdd2)= [3]
```

4. cartesian(other)

- Return the Cartesian product of this RDD and another one, that is, the RDD of all pairs of elements (a, b) where **a** is in **self** and **b** is in **other**.

```
In [17]: rdd2=sc.parallelize([1,1,2])
         rdd2=sc.parallelize(['a','b'])
         print('rdd1=',rdd1.collect())
         print('rdd2=',rdd2.collect())
         print('rdd1.cartesian(rdd2)=\n',rdd1.cartesian(rdd2).collect())
```

```
rdd1= [1, 1, 2, 3]
rdd2= ['a', 'b']
rdd1.cartesian(rdd2)=
 [(1, 'a'), (1, 'b'), (1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'), (3, 'b')]
```

1.3 Summary

- Chaining: creating a pipeline of RDD operations.
- counting, taking and sampling an RDD
- More Transformations: `filter, distinct, flatmap`
- Set transformations: `union, intersection, subtract, cartesian`

- See you next time!