spark-fundamentals

December 18, 2018

1 Spark Fundamentals

1.1 Working with Documentation

Being able to find and work with documentation is an essential skill for any programmer. Throughout this course there will be a number of exercises. Keep in mind that not only information from the course materials, but also all of the internet is available for you for reference.

For example, the PySpark API documentation can be found at **https://spark.apache.org/docs/latest/api/python**.

1.2 Our First Spark Program

In the following we are using the third-party package `findspark` to easily locate the Spark installation on our system:

```
In [1]: import findspark
        findspark.init()
```

Afterwards, we are able to import the `pyspark` module:

```
In [2]: import pyspark
```

Our first Spark program will be an approximation of the number π - not the most efficient one, but our program is going to use several fundamental features of Spark.

First, we create a `SparkContext` to communicate with the Spark cluster, and give our program a name:

```
In [3]: sc = pyspark.SparkContext(appName="Pi")
```

Now to the implementation of our algorithm: Here is the first function we need. It generates a random point in a square of size 1x1. (If you are wondering about the argument `x`, one argument is required for this to be used later as an argument to the `map` operation.)

```
In [4]: import random

In [5]: def random_point(x):
            return (random.random(), random.random())
```

We need to define one more function: The following function is given a point $p = (x, y)$ and checks whether $x^2 + y^2$ is less than 1:

```
In [6]: def inside(p):
            x, y = p
            return x*x + y*y < 1
```

The following code generates a large array of random points, distributes it to the cluster and calls our function once for each of them. It then counts the number of times our function returned `True` - which is approximately equal to π. Finally, it calls the `stop` method of the Spark context to terminate the program - this is necessary before we can create a new Spark context.

```
In [7]: %%time
        num_samples = 100000000
        count = sc.range(num_samples).map(random_point).filter(inside).count()
        pi = 4 * count / num_samples
        print(pi)
        sc.stop()
```

```
3.14205056
CPU times: user 20.7 ms, sys: 8.06 ms, total: 28.7 ms
Wall time: 41.8 s
```

(Curious why this works? Read more on how to calculate π via Monte Carlo approximation)

1.2.1 SparkContext and the Architecture of a Spark Program

Let us take apart the program step by step and look at each one. Our first step was to create a `SparkContext`.

```
In [8]: sc = pyspark.SparkContext(appName="Pi")
```

The `SparkContext` represents the connection to a Spark cluster and allows your Spark **driver application** to access the cluster. Each driver application commands a number of **executors**. These are processes that run on the nodes of the cluster and perform the actual computation and storage operations. They remain running as long as the driver application has a `SparkContext`. A program called the **cluster manager** sits between driver and executors and manages the resources given to the Spark program - for example, fairly assigning the number of nodes given to each program when multiple users are working on the cluster.

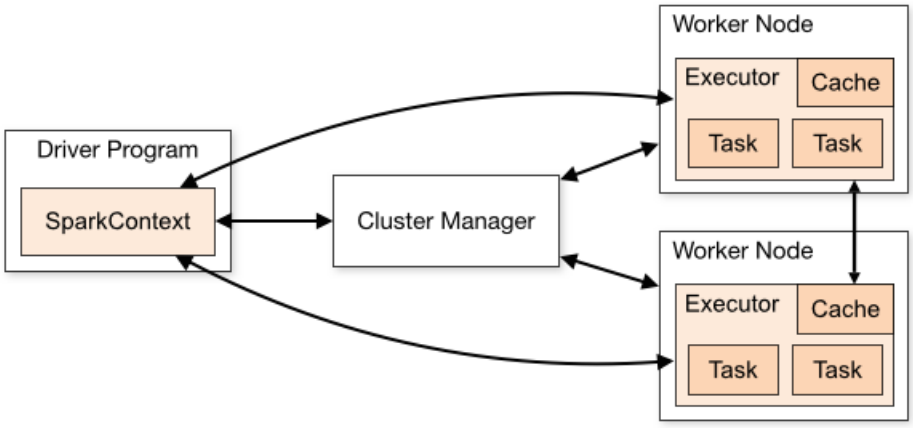
Source: MSDN

In our example program, we use the `SparkContext` to create a dataset of points in a parallel way and also distribute it to the executors. We start by creating a range of numbers up to the number of samples.

```
In [9]: %%time
        rdd = sc.range(num_samples)
```

```
CPU times: user 2.23 ms, sys: 1.82 ms, total: 4.05 ms
Wall time: 11.4 ms
```

With a call to `range`, we have created our first **Resilient Distributed Dataset** or **RDD**.



Architecture of a spark progarm

1.3 Resilient Distributed Datasets (RDDs)

The RDD is Spark's core data type, and it is in essence a *distributed collection of data objects*. It is *resilient* because it can cope with failure of some nodes of the cluster - and in a large distributed system, things like hardware failure become something that needs to be anticipated.

1.4 Transformations and Actions

Now to generate random points, we want to map each number from our range to a new random point:

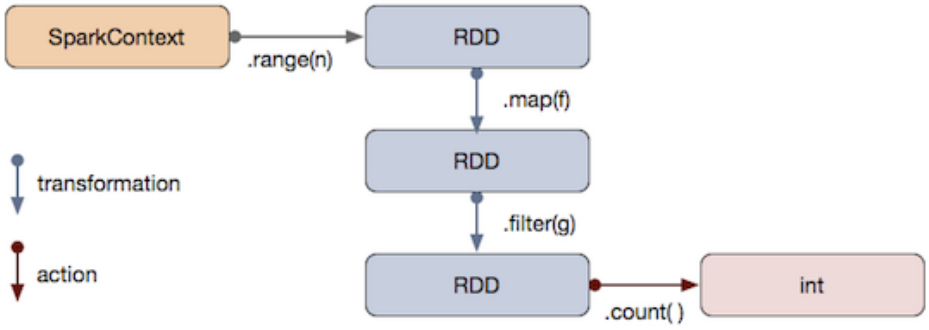
```
In [10]: %time
         rdd = rdd.map(random_point)

CPU times: user 4 µs, sys: 1e+03 ns, total: 5 µs
Wall time: 16.2 µs
```

In the next step, we want to keep only the points that pass our filter criterion defined in the `inside` function above:

```
In [11]: %%time
         rdd = rdd.filter(inside)

CPU times: user 24 µs, sys: 1e+03 ns, total: 25 µs
Wall time: 29.8 µs
```

1.4.1 Understanding Lazy Evaluation

Take a closer look at the running time measurement: Calling the `map` and `filter` methods on the RDD containing lots of random points was finished in a fraction of a second - this is because *no actual computation on the data has happened yet*. At this point, we need to understand the concept of **lazy evaluation** and learn about the difference between Spark **transformations** and **actions**.



operator graph

The `filter` method is a **transformation**. It is called on an RDD and returns another RDD. What actually happens under the hood when calling the method is not the computation of an RDD. When we call a transformation, we only add an operation to an **operator graph** (to be precise, a directed acyclic graph of RDDs and operations). This graph is an abstract description of the computation to be performed - basically, a roadmap of how an RDD is turned into another one. When working with big data, not computing immediately (**lazy evaluation**) has several advantages. Our computation might take very long, and we can first build a sequence of operations before triggering the computation and waiting for the result. Also (without going to much into the technical details of Spark), building the operator graph first allows Spark to analyze it before and coming up with an optimized plan for the actual computation (e.g. by combining operations that can be performed together, by copying or moving around as little data as possible, ...).

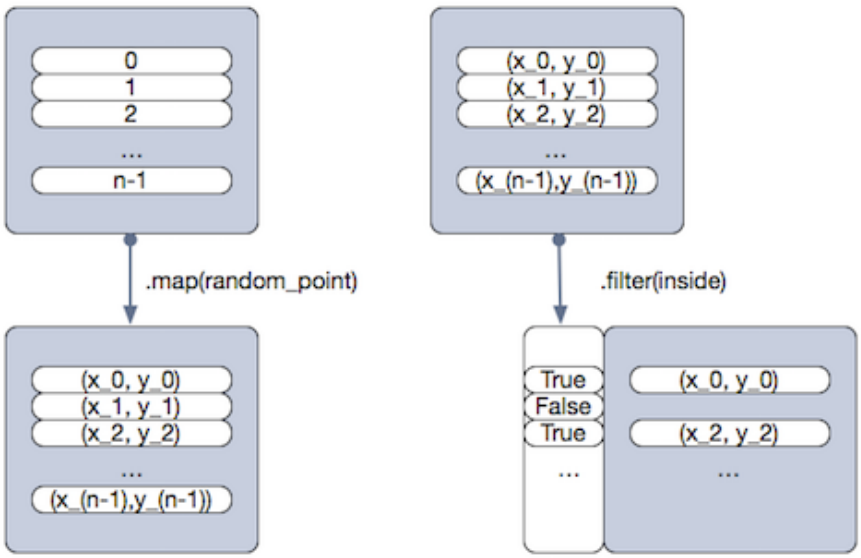
Let us take a closer look at the transformations `map` and `filter` and illustrate what they do with the elements of the RDD:

The last Spark operation in our program is a call to the `count` method of the RDD, which simply counts the number of elements in it. Watch the running time:

```
In [12]: %%time
         count = rdd.count()

CPU times: user 14.4 ms, sys: 4.94 ms, total: 19.4 ms
Wall time: 46.7 s
```

`count` is not a transformation but an **action**. When we invoke an action to retrieve a result, the *actual computation is triggered*. That is, the operator graph of transforms we have built before is turned into an execution plan, and the executors perform the computation on the nodes of the cluster. Then, a result is returned to the driver process.

```
In [13]: pi = 4 * count / num_samples
         print(pi)
         sc.stop()

3.14175424
```



map-filter

1.4.2 Common Transformations and Actions

Here we list some commonly used transformations and actions. Spark provides many more.

Transformations on a single RDD

method	purpose
`rdd.map(f)`	apply a function `f` to each element in the RDD and return an RDD of the result
`rdd.flatMap(f)`	apply a function `f` to each element in the RDD and return an RDD of the contents of the iterators returned
`rdd.filter(f)`	return an RDD consisting of only elements that pass the condition, i.e. for which function `f` returns `True`
`rdd.distinct()`	return RDD with duplicates removed
`rdd.sample(withReplacement, fraction, [seed])`	sample an RDD, with or without replacement

Transformations on two RDDs

method	purpose
`rdd.union(other_rdd)`	produce an RDD containing elements from both RDDs
`rdd.intersection(other_rdd)`	produce an RDD containing elements in the intersection of both RDDs
`rdd.subtract(other_rdd)`	produce RDD with the contents of the other RDD removed from the first one

Actions on an RDD

```
In [15]: sc.stop()
```

1.4.3 Hint: Write Anonymous Functions with Lambda Expressions

We have already seen how to write functions in Python with the `def` statement. Here is another way that is very frequently used in the context of Spark: Lambda expressions. Using the `lambda` keyword allows us to write down a function in a very concise way, in a single line. Lambda expressions are a key feature of the **functional programming** style.

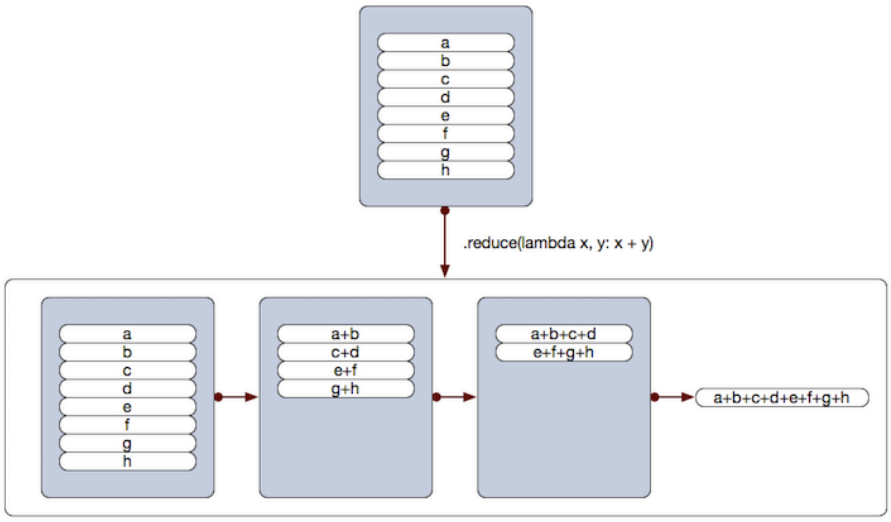
```
In [33]: def concat(a, b):
             """Concatenate two strings"""
             return f"{a}{b}"

In [34]: def apply(f, args):
             """Apply any function to the given arguments"""
             print(f"Applying {f} to {args}")
             return f(*args)

In [35]: concat("Hello ", "world")

Out[35]: 'Hello world'

In [36]: apply(concat, ("Hello ", "world"))

Applying <function concat at 0x10f9e09d8> to ('Hello ', 'world')


Out[36]: 'Hello world'

In [37]: apply(f=lambda a,b: f"{a}{b}", args=("Hello ", "world"))

Applying <function <lambda> at 0x10f9e0400> to ('Hello ', 'world')


Out[37]: 'Hello world'
```

1.4.4 Exercise: MapReduce

You may have heard of MapReduce in the context of big data. The name MapReduce originally referred to the proprietary technology of Google, where it was pioneered, but today it refers generally to a certain **programming model** for processing big data sets with a parallel, distributed algorithm on a cluster. As the name says, it relies on two main operations, `map` and `reduce`. And as you have seen, both are available in Spark: Sparks programming model includes MapReduce as part of its much larger set of transformations and actions.

The `reduce` operation is an action that iteratively combines (=aggregates) pairs of elements, two by two, until a single element is obtained. How to combine two elements is defined by the function given as arguments, that needs to be of the form $f(x, y) \to z$. `reduceByKey` is a similar action, but works only on RDDs containing key-value-pairs, and aggregates the values with the same key.

As an exercise, try impelemting our π approximation algorithm with `map` and `reduce` instead of `filter` and `count`.



reduce

```
In [21]: # once again, the original implementation with filter and count
         sc = pyspark.SparkContext(appName="Pi")
         count = sc.range(num_samples).map(random_point).filter(inside).count()
         pi = 4 * count / num_samples
         print(pi)
         sc.stop()
```

```
3.14215472
```

```
In [22]: # EXERCISE: approximation of pi using only range, map and reduce
```

1.5 Caching

Intermediate RDDs in the operator graph of our program are computed on demand when asking for a result via an action. In our example program, the filtered version of the RDD (containing only the points for which the `inside` function has returned `True`) is recomputed every time we call `count`:

```
In [23]: sc = pyspark.SparkContext(appName="PiCached")
```

```
In [24]: %%time
         count = sc.range(num_samples).map(random_point).filter(inside).count()
```

```
CPU times: user 12.3 ms, sys: 4.98 ms, total: 17.3 ms
Wall time: 43.5 s
```

```
In [25]: %%time
         count = sc.range(num_samples).map(random_point).filter(inside).count()
```

```
CPU times: user 13.4 ms, sys: 5.89 ms, total: 19.3 ms
Wall time: 40.1 s
```

This is often efficient when working with big data. However, in some cases (think of an iterative algorithm that needs to go over the intermediate RDD many times), it is more efficient to store an intermediate RDD rather than recomputing it. This can be done via Spark's functionality for **caching**. Here, we try to cache the RDD containing the points, then filtering and counting twice:

```
In [26]: cached_rdd = sc.range(num_samples).map(random_point).cache()

In [27]: %%time
         count = cached_rdd.filter(inside).count()

CPU times: user 13 ms, sys: 3.46 ms, total: 16.5 ms
Wall time: 1min 10s


In [28]: %%time
         count = cached_rdd.filter(inside).count()

CPU times: user 12 ms, sys: 5.31 ms, total: 17.3 ms
Wall time: 1min 11s


In [29]: pi = 4 * count / num_samples
         print(pi)
         sc.stop()

3.14109072
```

1.6 Exercise: Word Count

No Spark course is complete without a word count example. However, our approach to it is DIY: It's your turn to use Spark to count the words in the given text and output the 10 most frequent words in descending order.

```
In [38]: text_path = "../.assets/data/iliad/iliad.txt"

In [39]: !ls -lah {text_path}
         !head {text_path}

-rw-r--r--@ 1 cls  staff   1.1M May 27 21:57 ../.assets/data/iliad/iliad.txt
The Project Gutenberg EBook of The Iliad of Homer by Homer



This eBook is for the use of anyone anywhere at no cost and with almost no
restrictions whatsoever. You may copy it, give it away or re-use it under
```

```
the terms of the Project Gutenberg License included with this eBook or
online at http://www.gutenberg.org/license
```

```
In [32]: # your turn - DIY word count in Spark
         sc = pyspark.SparkContext(appName="WordCount")
         text_file = sc.textFile(text_path)
         # TODO:
         sc.stop()
```
