Spark Basics 1

This notebook introduces two fundamental objects in Spark:

- The Spark Context
- The Resilient Distributed DataSet or RDD

Spark Context

We start by creating a **SparkContext** object named **sc**. In this case we create a spark context that uses 4 *executors* (one per core)

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

```
In [2]: from pyspark import SparkContext
        sc = SparkContext(master="local[4]")
        sc

Out[2]: <pyspark.context.SparkContext at 0x110f4f250>
```

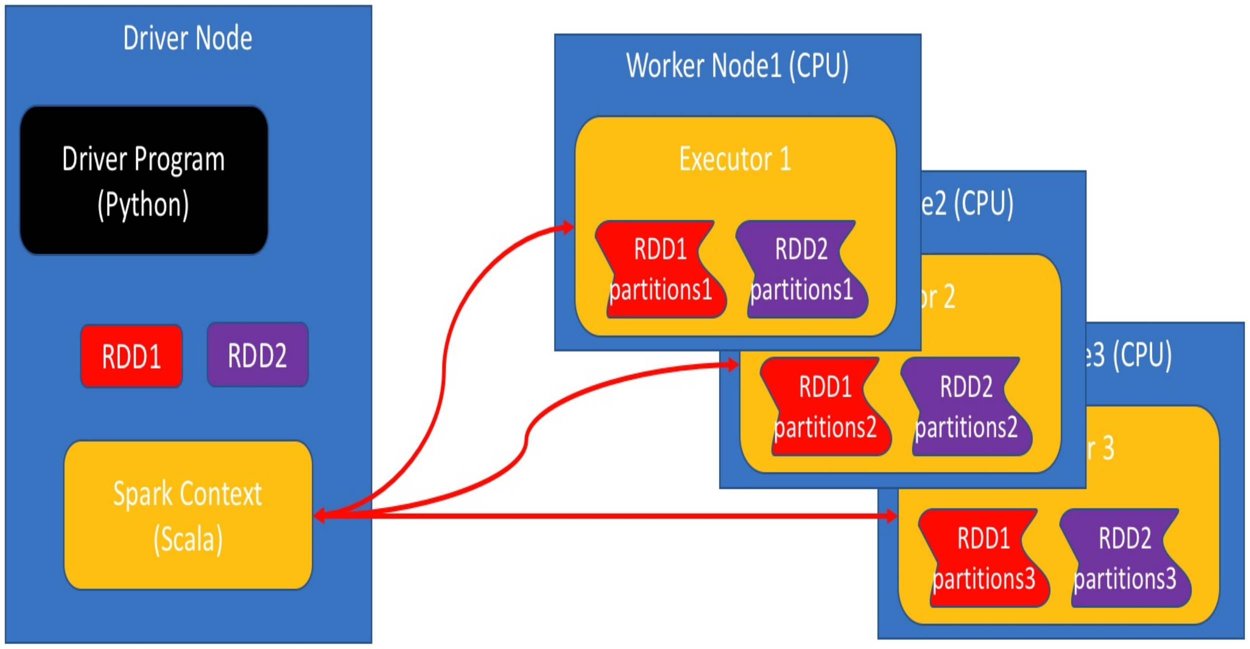
Only one sparkContext at a time!

When you run spark in local mode, you can have only a single context at a time. Therefor, if you want to use spark in a second notebook, you should first stop the one you are using here. This is what the method `.stop()` is for.

```
In [2]: # sc.stop() #commented out so that you don't stop your context by mistake
```

RDDs

RDD (or Resilient Distributed DataSet) is the main novel data structure in Spark. You can think of it as a list whose elements are stored on several computers.



The elements of each `RDD` are distributed across the **worker nodes** which are the nodes that perform the actual computations. This notebook, however, is running on the **Driver node**. As the RDD is not stored on the driver-node you cannot access it directly. The variable name `RDD` is really just a pointer to a python object which holds the information regardnig the actual location of the elements.

Some basic RDD commands

Parallelize

- Simplest way to create an RDD.
- The method `A=sc.parallelize(L)`, creates an RDD named `A` from list `L`.
- `A` is an RDD of type `ParallelCollectionRDD`.

In [8]:
```
A=sc.parallelize(range(3))
print type(A),A
```

```
<class 'pyspark.rdd.RDD'> ParallelCollectionRDD[4] at parallelize at Pyth
onRDD.scala:475
```

Collect

- RDD content is distributed among all executors.
- `collect()` is the inverse of `parallelize()'
- collects the elements of the RDD
- Returns a list

In [4]:
```
L=A.collect()
print type(L)
print L
```

```
<type 'list'>
[0, 1, 2]
```

Using `.collect()` eliminates the benefits of parallelism

It is often tempting to `.collect()` and RDD, make it into a list, and then process the list using standard python. However, note that this means that you are using only the head node to perform the computation which means that you are not getting any benefit from spark.

Using RDD operations, as described below, **will** make use of all of the computers at your disposal.

Map

- applies a given operation to each element of an RDD
- parameter is the function defining the operation.
- returns a new RDD.
- Operation performed in parallel on all executors.
- Each executor operates on the data **local** to it.

In [9]:
```
A.map(lambda x: x*x).collect()
```

Out[9]:
```
[0, 1, 4]
```

Note: Here we are using **lambda** functions, later we will see that regular functions can also be used.

For more on lambda function see here (http://www.secnetix.de/olli/Python/lambda_functions.hawk)

Reduce

- Takes RDD as input, returns a single value.
- **Reduce operator** takes **two** elements as input returns **one** as output.
- Repeatedly applies a **reduce operator**
- Each executor reduces the data local to it.
- The results from all executors are combined.

The simplest example of a 2-to-1 operation is the sum:

In [10]:
```
A.reduce(lambda x,y:x+y)
```

Out[10]:
```
3
```

Here is an example of a reduce operation that finds the shortest string in an RDD of strings.

In [11]:
```
words=['this','is','the','best','mac','ever']
wordRDD=sc.parallelize(words)
wordRDD.reduce(lambda w,v: w if len(w)<len(v) else v)
```

Out[11]:
```
'is'
```

Properties of reduce operations

- Reduce operations **must not depend on the order**
  - Order of operands should not matter
  - Order of application of reduce operator should not matter
- Multiplication and summation are good:

```
1 + 3 + 5 + 2                    5 + 3 + 1 + 2
```

- Division and subtraction are bad:

```
1 - 3 - 5 - 2                    1 - 3 - 5 -
2
```

Why must reordering not change the result?

You can think about the reduce operation as a binary tree where the leaves are the elements of the list and the root is the final result. Each triplet of the form (parent, child1, child2) corresponds to a single application of the reduce function.

The order in which the reduce operation is applied is **determined at run time** and depends on how the RDD is partitioned across the cluster. There are many different orders to apply the reduce operation.

If we want the input RDD to uniquely determine the reduced value **all evaluation orders must must yield the same final result**. In addition, the order of the elements in the list must not change the result. In particular, reversing the order of the operands in a reduce function must not change the outcome.

For example the arithmetic operations multiply `*` and add `+` can be used in a reduce, but the operations subtract `-` and divide `/` should not.

Doing so will not raise an error, but the result is unpredictable.

In [12]:
```
B=sc.parallelize([1,3,5,2])
B.reduce(lambda x,y: x-y)
```

Out[12]:
```
-9
```

Which of these the following orders was executed?

- $$((1-3)-5)-2$$

or

- $$(1-3)-(5-2)$$

Using regular functions instead of lambda functions

- lambda function are short and sweet.
- but sometimes it's hard to use just one line.
- We can use full-fledged functions instead.

In [13]:
```
A.reduce(lambda x,y: x+y)
```

Out[13]:
```
3
```

Suppose we want to find the

- last word in a lexicographical order
- among
- the longest words in the list.

We could achieve that as follows

In [14]:
```
def largerThan(x,y):
    if len(x)>len(y): return x
    elif len(y)>len(x): return y
    else:  #lengths are equal, compare lexicographically
        if x>y:
            return x
        else:
            return y

wordRDD.reduce(largerThan)
```

Out[14]:
```
'this'
```

In [ ]: