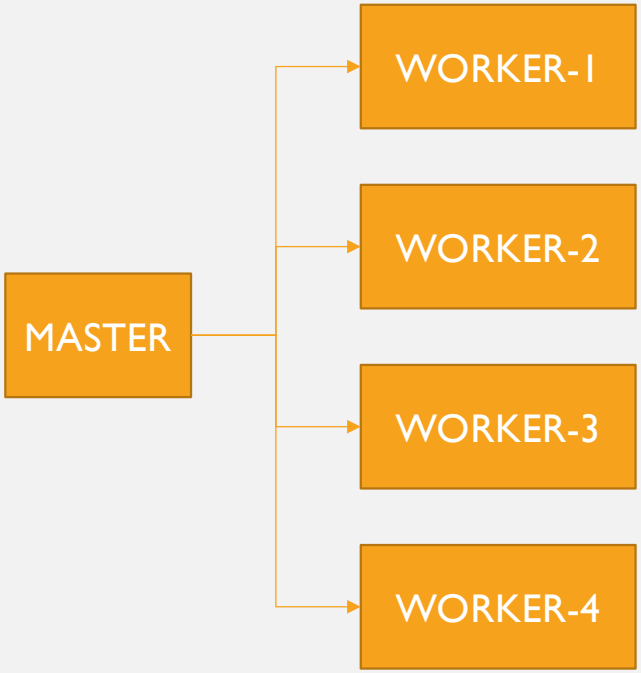
DDAM, 2019
DATA ANALYSIS WITH SPARK

Patrizio Dazzi

SPARK ARCHITECTURE

- 1 master, 4 workers



PYSPARK SHELL

- MASTER=spark://<host>:<port> ./bin/pyspark
  - E.g. MASTER=spark://localhost:7077 ./bin/pyspark

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.3.2
      /_/

Using Python version 3.7.4 (default, Aug 13 2019 15:17:50)
SparkSession available as 'spark'.
>>>
```

CREATE THE RDD

```
>>> data = range(20)
>>> myrdd = sc.parallelize(data)
>>> data
range(0, 20)

>>> myrdd
PythonRDD[1] at RDD at PythonRDD.scala:52
```

TRANSFORMATIONS

- transformations return a new RDD
- transformations are not executed immediately, but only when Sparks considers it needed. (This is called lazy evaluation, do you remember ?).

MAP

- Returns a new RDD by applying a given function.
- Let's compute the square of each value in myrdd:

```
>>> def sq(x): return x*x
>>> squared = myrdd.map(sq)
>>> print (squared.collect())
[Stage 0:>                                                          (0 + 24) / 24]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361]
```

USING LAMBDA FUNCTIONS

- A lambda function is a small anonymous function, i.e. that is not bound to an identifier.
- A lambda function can take any number of arguments, but can only have one expression.
- Lambda functions are often arguments being passed to higher-order functions
- Syntax: lambda *arguments* : *expression*

```
x = lambda a, b : a * b
print(x(5, 6))
```

- With Pyspark:

```
>>> squared = myrdd.map(lambda x: x*x)
>>> print (squared.collect())
```

FLATMAP (1)

- map(func): Return a new distributed dataset formed by passing each element of the source through a function func.
- flatMap(func): 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).
- You can say for each input value, the flatMap outputs a sequence which can have 0 or more elements which are flattened to form output RDD.
- Useful when the map function generates more then one output.

FLATMAP (2)

```
>>> withmap = myrdd.map(lambda x: (x*x, x*x*x))

>>> print (withmap.collect())

[(0, 0), (1, 1), (4, 8), (9, 27), (16, 64), (25, 125), (36, 216),
(49, 343), (64, 512), (81, 729), (100, 1000), (121, 1331), (144,
1728), (169, 2197), (196, 2744), (225, 3375), (256, 4096), (289,
4913), (324, 5832), (361, 6859)]


>>> withflatmap = myrdd.flatMap(lambda x: (x*x, x*x*x))

>>> print withflatmap.collect()

[0, 0, 1, 1, 4, 8, 9, 27, 16, 64, 25, 125, 36, 216, 49, 343, 64,
512, 81, 729, 100, 1000, 121, 1331, 144, 1728, 169, 2197, 196,
2744, 225, 3375, 256, 4096, 289, 4913, 324, 5832, 361, 6859]
```

FLAT MAP – TEXT PROCESSING

- This is useful with text processing. Let's find the words of the following three lines of text.

```
>>> divina = [ "Nel mezzo del cammin di nostra vita", "mi ritrovai per una selva oscura", "ché la diritta via era smarrita."]

>>> divinardd = sc.parallelize(divina)

>>> words = divinardd.flatMap(lambda x:x.split())

>>> print words.collect()

['Nel', 'mezzo', 'del', 'cammin', 'di', 'nostra', 'vita', 'mi', 'ritrovai', 'per', 'una', 'selva', 'oscura', 'ché', 'la', 'diritta', 'via', 'era', 'smarrita.']
```

WITH MAP…

- … let's see the behaviour

```
>>> wordsm = divinardd.map(lambda x:x.split())
>>> print (wordsm.collect())
[['Nel', 'mezzo', 'del', 'cammin', 'di', 'nostra',
'vita'], ['mi', 'ritrovai', 'per', 'una', 'selva',
'oscura'], ['ché', 'la', 'diritta', 'via', 'era',
'smarrita.']]
```

FILTER AND SAMPLE

- Filter select a subset of items

```
>>> even = myrdd.filter(lambda x: x%2==0)
>>> print(even.collect())
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

- Sample draw a random sample of the data, with or without replacement. Let's take 20% of the data.

```
>>> sample = myrdd.sample(False, 0.20) # false means without replacement
>>> print (sample.collect())
[2, 4, 13, 15]
```

DISTINCT

- Remove duplicates. Let's try on a toy dataset.

```
>>> distinct =
sc.parallelize([1,2,2,3,3,3,4,4,4,4]).distinct()

>>> print distinct.collect()
```

[4, 1, 2, 3]

UNION AND INTERSECTION

- Union

```
>>> myrdd2 = sc.parallelize(range(10, 30))
>>> union = myrdd.union(myrdd2)
>>> print(union.collect())
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 10,
11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
```

- Intersection

```
>>> intersection = myrdd.intersection(myrdd2)
>>> print(intersection.collect())
[16, 17, 10, 18, 11, 19, 12, 13, 14, 15]
```

SUBTRACTION AND CARTESIAN PRODUCT

- Subtraction

```
>>> subtraction = myrdd.subtract(myrdd2)
>>> print(subtraction.collect())
[0, 8, 1, 9, 2, 3, 4, 5, 6, 7]
```

- Cartesian (every pair-wise combination).

```
>>> cartesian = myrdd.cartesian(myrdd2)
>>> print(cartesian.collect())
[(0, 10), (0, 11), (0, 12), (0, 13), …, (19, 29)]
```

JOINS (STILL TRANSFORMATIONS)

- Keys can be used for database-like join operation with the usual semantic:
  - join : performs the usual join based on keys
  - rightOuterJoin and leftOuterJoin : allows for missing values

```
>>> left = sc.parallelize( [(1, "red"), (3, "blue"), (3, "green")])
>>> right = sc.parallelize( [(3, "apples")])
>>> left.join(right).collect()
[(3, ('blue', 'apples')), (3, ('green', 'apples'))]
```

OTHER TRANSFORMATIONS

- keys : returns the list of keys
- values : returns the list of values
- mapValues and flatMapValues : applies the given function to values leaving keys unchanged.
- sortByKey : sort the list by key
- groupByKey : creates pairs key and list of values associated to the key.
- combineBy : similar to aggregate

ACTIONS

- Actions are the operations that return a final value to the driver program or write data to an external storage system.
- For this reason, the Spark computation is actually triggered when an action is invoked.
- The one we used for the previous example is the collect: Returns, or better materializes the RDD at the driver program.
- The collect action returns the whole RDD to the driver program. The RDD is potentially very large and the data transfer may be very expensive.

```
>>> print myrdd.collect()
```

COUNT AND COUNT BY VALUE

- Count returns the number of elements in the RDD.

```
>> print(myrdd.cartesian(myrdd2).count())
400
```

- Count by value returns the number of occurrences of each distinct value in the RDD.

```
>>> print(squared.countByValue())
```

- defaultdict(<class 'int'>, {0: 1, 1: 1, 4: 1, 9: 1, 16: 1, 25: 1, 36: 1, 49: 1, 64: 1, 81: 1, 100: 1, 121: 1, 144: 1, 169: 1, 196: 1, 225: 1, 256: 1, 289: 1, 324: 1, 361: 1})

ACTIONS ON KEY/VALUE PAIRS RDD

Three additional functions are made available for Key/Value pairs RDD.
Count by Key:
>>> left.countByKey()
defaultdict(<type 'int'>, {1: 1, 3: 2})

Look up:
>>> left.lookup(3)
['blue', 'green']

Collect as Map: It is possible to materialize the RDD at the driver as a dictionary. >>> mymap = left.groupByKey().collectAsMap()
>>> for key in mymap:
>>> print key, list(mymap[key]) # this is to convert value from special spark type
1 ['red']
3 ['blue', 'green']

TAKE AND TOP

- Take a few elements from the RDD. It's not sorted and it is not random.

```
>>> print(myrdd.take(3))
```

[0, 1, 2]

- Top returns the top (largest) elements in descending order.

```
>>> print(myrdd.top(3))
```

[19, 18, 17]

REDUCE

- Reduce aggregates elements according to the provided reduce function.
- Any function can be used instead of sum (in the following example) with the following constraints. The function should accept two parameters and return the aggregated value. Function should be associative and distributive.

```
>>> def sum(x,y): return x+y
>>> myrdd.reduce( sum )
>>> myrdd.reduce( lambda x,y: x+y )
```

- Note: the reduce is executed in parallel, e.g., sum of the elements managed by each machine are computed independently and then merged. Therefore, there is no guarantee about the order of the operations on the RDD.
- It's important to remind that partial operations (e.g., sums) are executed and then merged.

AGGREGATE (1)

- Complex aggregation of the elements in the RDD. Useful when the kind of information to be aggregated is different from the kind of information in the RDD.
- It introduces the concept of accumulators, i.e., the information to be aggregated. It takes 3 parameters:
  - the initial value of the accumulator a function that mergers an accumulator with a value in the RDD a function that merges two accumulators.
- Let's compute the average of the values in a given RDD. We use a vector of two positions as accumulator, where the first position stores the sum of the elements and the second stores the number of elements. Eventually, the two values in the accumulator are used to compute the average.

AGGREGATE (2)

```
# empty accumulator ( partial_sum, partial_count )
>>> empty_acc = (0.0, 0.0)

# merge by summing partial_sum and adding 1 to partial_count
>>> def mergeValue(acc, value): return (acc[0] + value, acc[1] + 1)

# merge by summing partial_sums and partial_counts
>>> def mergeAccum(acc1, acc2): return (acc1[0] + acc2[0], acc1[1] + acc2[1])

# spark aggregate
>>> sum_and_count = myrdd.aggregate( empty_acc, mergeValue, mergeAccum )
>>> print ("Average is", sum_and_count[0]/sum_and_count[1])
Average is 9.5
```

FOREACH

- Applies a given function to each element of the RDD. This is different from map as it does not create a new RDD but it actually generates actions.

```
>>> def f(x): print ("This is x:", x)
>>> myrdd.foreach( f )
This is x: 10
…
This is x: 19
This is x: 0
…
This is x: 9
```