Big Data analysis with Map-Reduce

DSC 2023, November, Belgrade

Ana Vranic
Scientific Computing Laboratory, Institute of Physics Belgrade, Serbia,
Social Physics and Complexity Laboratory, LIP, Portugal

Table of contents

1. Introduction

In Data Science, we often deal with big amounts of data. In those cases, many standard approaches won't work as expected, and to process big data, we need to apply a different technique called MapReduce. The main problem when dealing with big data is that the data size is so large that filesystem access times become a dominant factor in the execution time. Because of that, it is not efficient to process big data on a standard MPI cluster machines. With distributed computing solutions like Hadoop and Spark clusters, which rely on the MapReduce approach, big volumes of data are processed and created by diving work into independent tasks, performing the job in parallel. For the first time, the MapReduce approach was formalized by Google, in the paper MapReduce: Simplified Data Processing on Large Clusters when they encountered a problem in indexing all pages on the WWW.

map() and reduce() in python

Before we further explain the MapReduce approach, we will review two functions from Python, `map()` and `reduce()`, introduced in functional programming. In imperative programming, computation is carried through statements, where the execution of code changes the state of variables. Unlike in functional programming, the state is no longer necessary because functional programming works on immutable data types; functions create and work on new data sets while the original dataset is intact.

- map() applies the function to each element in the sequence and returns the resulting sequence. map() also returns the memory address of the returned map generator object and has to be called with a list() or through the loop.

```
ls = list(range(10))
list(map(lambda x: x**2, ls))
```

```
[1, 1, 4, 9, 16, 25, 36]
```

- reduce() function returns a single value. It applies the function to the sequence elements from left to right.

```
from functools import reduce
ls = list(range(1, 10, 2))
reduce(lambda x, y: x*y, ls) #(1*3*5*7*9)
```

```
945
```

1.1 Word Count in python

Counting the number of words in the document is the simplest example when learning the MapReduce technique. In this tutorial, we will work on the Moby Dick book. In python, we can use the following code:

word_count.py

```
import re
WORD_REGEX = re.compile(r"[\w]+")

# remove any non-words and split lines into separate words
# finally, convert all words to lowercase

def splitter(line):
    line = WORD_REGEX.findall(line)
    return map(str.lower, line)


sums = {}
try:
    in_file = open('pg2701.txt', 'r')

    for line in in_file:
        for word in splitter(line):
            sums[word] = sums.get(word, 0) + 1

    in_file.close()

except IOError:
    print("error performing file operation")
else:
```

```
    M = max([x for x in sums], key=lambda k: sums[k])
    print("max: %s = %d" % (M, sums[M]))
```

Ater running the program we will get:

```
max: the = 14620
```

A program written like this runs only on one processor, and we expect that the time necessary to process the whole text is proportional to the size of the text. Also, as the size of the dictionary grows, the performance degrades. When the size of the dictionary reaches the size of RAM or even swap space, the program will be stopped.

In the map-reduce approach, we can avoid memory issues we may encounter. Those aproaches are scalable, and can be tested on the smaller datasets on local machine. When data becomes big, we'll need to process and store data in distributed frameworks like Hadoop or Spark or online computing services like AWS or Azure, where we can use the same principles and patterns we will learn in this tutorial.

1.2 Map-Reduce

MapReduce consists of 3 steps:

- Map step which produces the intermediate results
- Shuffle step, which groups intermediate results with the same output key
- Reducing step that processes groups of intermediate results with the same key

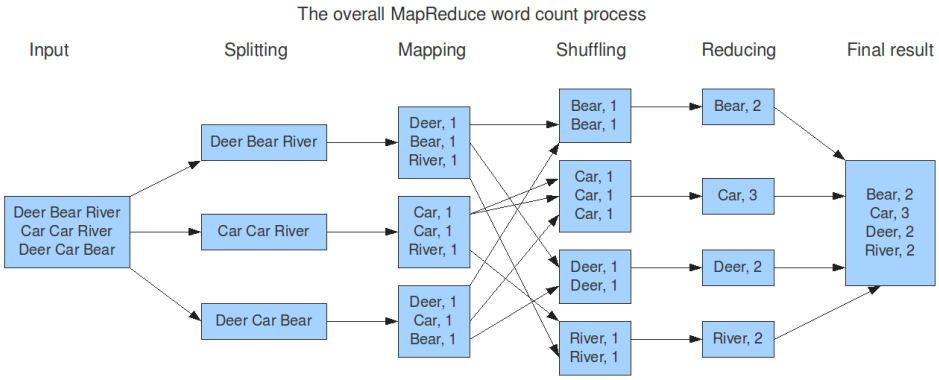


Figure credits: New York University

This approach works on data sets that consist of data records. The idea is to define the calculation in terms of two steps - a map step and a reduce step. The map operation is applied to every data record in the set and returns a list of key-value pairs. Those pairs are then collected, sorted by key, and passed into the reduce operation, which takes a key and a list of values associated with that key and then computes the final result for that key, returning it as a key-value pair.

2. Hadoop

Apache Hadoop is an open-source implementation of a distributed MapReduce system. A Hadoop cluster consists of a name node and a number of data nodes. The name node holds the distributed file system metadata and layout, and it organizes executions of jobs on the cluster. Data nodes hold the chunks of actual data and execute jobs on their local subset of the data.

Hadoop was developed in Java, and its primary use is through Java API; however, Hadoop also offers a “streaming” API, which is more general and it can work with map-reduce jobs written in any language which can read data from standard input and return data to standard output. In this tutorial, we will provide examples in Python.

2.1 Python Streaming

If you prefer languages other than Java, Hadoop offers the streaming API. The term streaming here refers to how Hadoop uses standard input and output streams of your non-java mapper and reducer programs to pipe data between them. Relying on stdin and stdout enables easy integration with any other language.

In the first example, we will implement word count. The mapper is defined as following:

map.py

```
!/usr/bin/env python3
import sys
import re

WORD_REGEX = re.compile(r"[\w]+")

for line in sys.stdin:
    for word in WORD_REGEX.findall(line):
        print( word.lower(), 1)
```

This mapper will read text from standard input, and for each word, it will echo that word and the number 1 as the number of times the word appeared. The reducer will take those key-value pairs and sum the number of appearances for each word:

reduce.py

```
#!/usr/bin/env python3
import sys

prevWord = ''
prevCount = 0
for line in sys.stdin:
    word, count = line.split()
    count = int(count)
    if word == prevWord:
        prevCount += 1
        continue
    else:
        if prevWord != '':
            print(prevWord, prevCount)
        prevWord = word
        prevCount = 1
if prevWord != '':
    print(prevWord, prevCount)
```

The reducer is more complicated as it must keep track of which key it is reducing on. This happens because it will get all the values for a single key on a separate key-value line, i.e. hadoop streaming mechanism won’t automatically collect all the values for a key into an array of values.

If we want to test our code locally, on the Mobi Dick book, it can be run as:

```
cat pg2701.txt | python map.py | sort | python reduce.py >  word_count.dat
```

When moving to the Hadoop cluster, the scripts map.py and reduce.py stay the same, and we can run it as.

```
$ mapred streaming  \
-input "2701.txt" \
-output "word_count.dat" \
-mapper mapper.py -reducer reducer.py \
-file mapper.py -file reducer.py
```

The input and output parameters specify the locations for input and output data on the HDFS file system - Hierarchical distributed file system. Mapper and reducer parameters specify the mapper and reducer programs, respectively. The following parameters specify files on the local file system that will be uploaded to Hadoop and made available in the context of that job. Here, we define our python scripts to make them available for execution on the data nodes.

2.2 Calculating ELO ratings

We can download data from repository

```
wget "https://github.com/JeffSackmann/tennis_wta/archive/refs/heads/master.zip"
```

The dataset contains `.csv files` with wta_matches from 1968 until 2023.

For given tennis data over several years of matches, we will calculate the ELO ratings of each player, using MapReduce, which should reflect the player's relative skills. After each match, we update the ratings of players. The ELO rating of player A is updated by formula:

\[R^{'}_a = R_a + K(S_a-E_a) \]

- \(R^{'}_a\) is new rating,
- \(R_a\) is previous rating,
- \(S_a\) is actual outcome of the match. The actual outcome of one player may be victory \(S_a=1\) or loss \(S_a=0\) so \(0\leq S_a \leq 1\).
- \(E_a\) is expected outcome of the match while. To map the expectation of the outcome from 0 to 1, we can use logistic curve, so \(E_a = Q_a / (Q_a+Q_b)\), where \(Q_a = 10^{R_a/c}\), \(Q_b = 10^{R_b/c}\). The factor c can be 400.
- Parameter K is scaling factor, which determines how much influence each match can have, and in this example can be set on \(100\) .

The mapper:

Each line in data contains attributes about the match such as winner, loser, surface. To select these elements, we need to split each line on comas, and select 2nd, 10th and 20th position. And we'll pass these features in the key-value pairs into json string, and result can be printed to standard output using json.dumps function from the JSON module.

Mapper for analyzing tennis score

elo_map.py

```
#!/usr/bin/python3
import json
from sys import stdin

def clean_match(match):
    ms = match.split(',')
    match_data = {'winner': ms[10],
                  'loser':ms[18],
                   'surface': ms[2]
                  }
    return match_data

if __name__=="__main__":
    for line in stdin:
        print(json.dumps(clean_match(line)))
```

The reducer:

The reducer takes JSON objects, from standard input. In elo_acc function for each match we update the ratings of users and store them into dictionary acc. Also the obtained values are round to 5 decimals using round5 function.

elo_reduce.py

```
#!/usr/bin/python3
import json
from sys import stdin
from functools import reduce

def round5(x):
```

```
    return 5*int(x/5)

def elo_acc(acc, nxt):

    match_info = json.loads(nxt)
    w_elo = acc.get(match_info['winner'], 1400)
    l_elo = acc.get(match_info['loser'], 1400)
    Qw = 10**(w_elo/400)
    Ql = 10**(l_elo/400)
    Qt = Qw+Ql

    acc[match_info['winner']] = round5(w_elo + 100*(1-(Qw/Qt)))
    acc[match_info['loser']] = round5(l_elo - 100*(Ql/Qt))
    return acc

if __name__=="__main__":
    # return dictionary
    xs = reduce(elo_acc, stdin, {})
    topN = (sorted(xs.items(), key=lambda item: item[1], reverse=True))[:20]

    for player, rtg in topN:
        print(rtg, player)
```

We can always check the output of our scripts in local:

```
cat tennis_wta-master/wta_matches_* | python elo_map.py | sort | python elo_reduce.py
```

after running script we'll get result similar to this:

```
3075 Zina Garrison
2930 Victoria Azarenka
2845 Zarina Diyas
2765 Zuzana Ondraskova
2745 Yung Jan Chan
2745 Yvonne Vermaak
2710 Tatiana Golovin
2700 Yurika Sema
...
```

To run it on Hadoop cluster:

```
$ mapred streaming \
-file ./elo-mapper.py -mapper ./elo-mapper.py \
-file ./elo-reducer.py -reducer ./elo-reducer.py \
-input '${path}/wta_matches_200*.csv' \
-output "tennis_ratings"
```

3. mrjob

mrjob is a Python MapReduce library that wraps Hadoop streaming and allows us to write the MapReduce programs in a more Pythonic manner. With mrjob, it is possible to write multistep jobs. Programs can be tested locally, run on the Hadoop cluster, and run in the Amazon cloud using Amazon Elastic MapReduce (EMR).

- instalation `$ pip install mrjob`

In mrjob, the MapReduce function is defined as class MRClass, which contains the methods that define the MapReduce job:

- the mapper() defines the mapper. It takes (key, values) as arguments and yields tuppels (output_key, output_valies)
- the combiner() defines the process that runs after the mapper and before the reducer. It receives all data from the mapper, and the output of the paper is sent to the reducer. The combiner's input is the key, yielded by the mapper, and a value, which is a generator that yields all values yielded by one mapper that corresponds to the key. The combiner yields tuples of (output_key, output_value) as output.
- the reducer() defines the reducer for the MapReduce job. It takes a key and an iterator of values as arguments and yields tuples of (outup_key, output_value)
- The final component enables the execution of mrjob.

```
if __name__ == '__main__':

    MRClass.run()
```

3.1 Word count example with mrjob

We will perform a word count on Moby Dick book downloaded from project Gutenberg

```
wget "https://gutenberg.org/cache/epub/2701/pg2701.txt"
```

mrjob script wordcount_mrjob.py:

mrjob_wc.py

```
from mrjob.job import MRJob
import re

WORD_REGEX = re.compile(r"[\w]+")
class MRWordCount(MRJob):

    def mapper(self, _, line):
        for word in WORD_REGEX.findall(line):
            yield word.lower(), 1


    def reducer(self, word, counts):
        yield(word, sum(counts))


if __name__ == '__main__':
    MRWordCount.run()
```

To run it localy:

```
$ python wordcount_mrjob.py 'pg2701.txt'
```

We can execute the mrjob locally: `$ python mrjob.py input.txt`. The mrjob writes output to stout. To save results to file we can run `$ python mrjob.py input.txt > out.txt` We can also pass the multiple files `$ python mrjob.py input.txt input2.txt input3.txt`.

Finally, with the `-runner/-r` option, we can define how the job executes. If the job executes in the Hadoop cluster `$ python mrjob.py -r hadoop input.txt` If we run it on the EMR cluster `$ python mrjob.py -r emr s3://input-bucket/input.txt`.

3.2 Chaining map-reduce

With mrjob, we can easily chain several map-reduce functions. For example, if we need to calculate the word with maximum frequency in the dataset. To do that, we need to override the steps() method. The code will have a mapper and reducer, the same as in the previous task. Then, the second mapper uses the reducer's output, which maps all (word, count) pairs to the same key, None. The shuffle step of map-reduce will collect them all into one list corresponding to the key None. Then reducer_post will sort the list of (word, word_count) pairs by word_count and yield the word with maximum frequency.

mrjob_wf.py

```
from mrjob.job import MRJob
from mrjob.step import MRStep
import re

WORD_REGEX = re.compile(r"[\w]+")

class MRMaxFreq(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper,
                   reducer=self.reducer),
            MRStep(mapper=self.mapper_2,
                   reducer=self.reducer_2)
        ]

    def mapper(self, _, line):
        for word in WORD_REGEX.findall(line):
            yield word.lower(), 1

    def reducer(self, word, counts):
        yield word, sum(counts)


    # keys: None, values: (word, word_count)
    def mapper_2(self, word, word_count):
        yield None, (word, word_count)

    # sort list of (word, word_count) by word_count
    def reducer_2(self, _, word_count_pairs):
        yield max(word_count_pairs, key=lambda p: p[1],)

if __name__ == "__main__":
    MRMaxFreq().run()
```

We run it as previous, additionaly the output can be redirected to the file "max_freq_word.txt"

```
$ python word_freq_mrjob.py 'pg20701.txt' > 'max_freq_word.txt'
$ cat 'max_freq_word.txt'
"the" 14620
```

3.3 Passing arguments to mrjob

In this example we will preprocess tennis dataset in order to check Williams sisters rivaly with MRJob. Here we will select matches when 'Serena Williams' and 'Venus Williams' played against eachother, and calculate how many times each sister won depending on the surface.

mrjob_williams.py

```
from mrjob.job import MRJob
from functools import reduce

def make_counts(acc, nxt):
    acc[nxt] = acc.get(nxt,0) + 1
    return acc

def my_freq(xs):
    return reduce(make_counts, xs, {})

class Williams(MRJob):

    def mapper(self, _, line):
        fields = line.split(',')
        players = [fields[10], fields[18]] #(winner, loser)
        if 'Serena Williams' in players and 'Venus Williams' in players:
            yield fields[2], fields[10]

    def reducer(self, surface, results):
        counts = my_freq(results)
        yield surface, counts
```

```
if __name__ == "__main__":
    Williams.run()
```

```
python mrjob_wiliams.py tennis_wta-master/wta_matches_*
```

Instead of overcoding the script with 'Serena Williams' and Venus Williams we can pass arguments to mrjob using `add_passthru_arg` option, and we will have to define new function `configure_args()` which loads passed arguments.

mrjob_2players.py

```
from mrjob.job import MRJob
from functools import reduce

def make_counts(acc, nxt):
    acc[nxt] = acc.get(nxt,0) + 1
    return acc

def my_freq(xs):
    return reduce(make_counts, xs, {})

class Williams(MRJob):

    def configure_args(self):
        super(Williams, self).configure_args()
        self.add_passthru_arg("-p1", "--player1", help="player1")
        self.add_passthru_arg("-p2", "--player2", help="player1")


    def mapper(self, _, line):
        fields = line.split(',')
        players = [fields[10], fields[18]]
        if self.options.player1 in players and self.options.player2 in players:
            yield fields[2], fields[10]

    def reducer(self, surface, results):
        counts = my_freq(results)
        yield surface, counts

if __name__ == "__main__":
    Williams.run()
```

```
python mrjob_2players.py tennis_wta-master/wta_matches_* --player1 "Serena Williams" --player2 "Venus Williams"
```

4. Spark

Apache Spark is another popular cluster computing framework for big data processing. Contrary to Hadoop, it takes advantage of high-RAM computing machines, which are now available. Spark processes data in memory on the distributed network instead of storing data in the filesystem. This can improve the processing time. Spark's advantages are natively supporting programming languages like Scala, Java, Python, and R. Spark has a direct Python interface - **pyspark**, which uses the same analogy of map and reduce. It can be used interactively from the command shell or jupyter notebook. Spark can query SQL databases directly, and DataFrame API is similar to pandas.

Installation of pyspark requires installed Java. From there, we can install pyspark as any other python package and test our scrips locally before moving to the Spark cluster.

```
pip install pyspark
```

Before we introduce the main features of the pyspark, let's see how we can write the simplest problem word count.

4.1 PySpark

Word Count in PySpark

word_count_spark.py

```
from pyspark import SparkContext

def main():
  sc = SparkContext(appName='SparkWordCount')
  input_file = sc.textFile('pg2701.txt')
  counts = input_file.flatMap(lambda line: line.split()) \
  .map(lambda word: (word, 1)) \
  .reduceByKey(lambda a, b: a + b)
  counts.saveAsTextFile('output')
  sc.stop()

if __name__ == '__main__':
  main()
```

The code can be run with `spark-submit word_count_spark.py` the output will appear in folder output/0001 and output/0002, depending on the number of processes.

The `sc = SparkContext(appName='SparkWordCount')` creates a context object, which tells Spark how to access the cluster. The `input_file = sc.textFile('pg2701.txt')` loads data. The third line performs multiple input data transformations, similar to before. Everything is automatically parallelized and runs across multiple nodes. The variable counts is not immidiately computed due to laziness of transformations. When we call function reduce(), which is action, Spark devides the computations into tasks on separate machines. Each machine runs the map and reduction on its local data, returning only the results to the driver program.

Resilient Distributed Datasets (RDDs)

RDDs are immutable collections of data distributed across machines, which enables operations to be performed in parallel. They can be created from collections by calling `parallelize()` method:

```
data = [1, 2, 3, 4, 5, 6]
rdd = sc.parallelize(data)
rdd.glom().collect()
```

```
[1, 2, 3, 4, 5, 6]
```

RDD.glom() returns a list of elements within each partition, while RDD.collect() collect all elements to the driver node. To specify the number of partitions:

```
rdd = sc.parallelize(data, 4)
rdd.glom().collect()
```

Another way to create Rdd is from a file using `textFile()` method, as we did in the word count example.

RDD Operations

Here are listed some commonly used operations that can be applied to the RDDs:

- map() map. function returns a RDD by applying function to each element of the source RDD

```
data = [1, 2, 3, 4, 5, 6]
rdd = sc.parallelize(data)
map_result = rdd.map(lambda x: x * 2)
map_result.collect()
```

[2, 4, 6, 8, 10, 12]

- flatMap() returns a flattened version of results.

```
data = [1, 2, 3, 4]
rdd = sc.parallelize(data)
rdd.map(lambda x: [x, pow(x,2)]).collect()
```

[[1, 1], [2, 4], [3, 9], [4, 16]]``

```
rdd = sc.parallelize([1, 2, 3, 4])
rdd.flatMap(lambda x: [x, pow(x,2)]).collect()
```

[1, 1, 2, 4, 3, 9, 4, 16]

- .groupBy()

```
rdd = sc.parallelize(["apple", "banana", "cantaloupe"])
xs = rdd.groupBy(lambda x: x[0]).collect()
print(sorted([(x, sorted(y)) for (x, y) in xs]))
```

[('a', ['apple']), ('b', ['banana']), ('c', ['cantaloupe'])]

- .groupByKey() python

```
rdd = sc.parallelize([("pet", "dog"), ("pet", "cat"),("farm", "horse"), ("farm", "cow")])
xs = rdd.groupByKey().collect()

[(x, list(y)) for (x, y) in xs]
```

[('farm', ['horse', 'cow']), ('pet', ['dog', 'cat'])]

- filter(func) returns a new RDD contains only elements that function return as true

```
rdd = sc.parallelize([1, 2, 3, 4, 5, 6])
rdd.filter(lambda x: x % 2 == 0).collect()
```

[2, 4, 6]

- distinct. this returns unique elements in the list

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

- reduce

```
rdd = sc.parallelize([1, 2, 3])
print(rdd.reduce(lambda a, b: a+b))
```

6

- reduceByKey

```
rdd = sc.parallelize([(1, 2), (1, 5), (2, 4)])
rdd.reduceByKey(lambda a, b: a+b).collect()
```

[(1, 7), (2, 4)]

4.2 Page-rank algorithm

The PageRank was used as Google's ranking system, resulting in websites with higher PageRank scores showing up higher in Google searches. PageRank can be performed on the graph (network) structured datasets. PageRank will rank nodes, giving the ranking of nodes by their influence. The more followers the node has, the more influential it is, and the more those followers are influential, the more they will contribute to the node's rank. More details about the PageRank algorithm can be found in the original paper.

Page rank can be calculated as:

$r_i = (1-d) + d(\sum_{j=1}^{N} I_{ij} \frac{r_j}{n_j})$

The PageRank of a node is (1-dumping factor) + every node that points to node i will contribute with page rank of node j/ number of outgoing links. The PageRank has analogy with random walker over internet. The probability that walker will follow link is proportionall to the damping factor d, while probability that walker jumps to any random page is $1-d$.

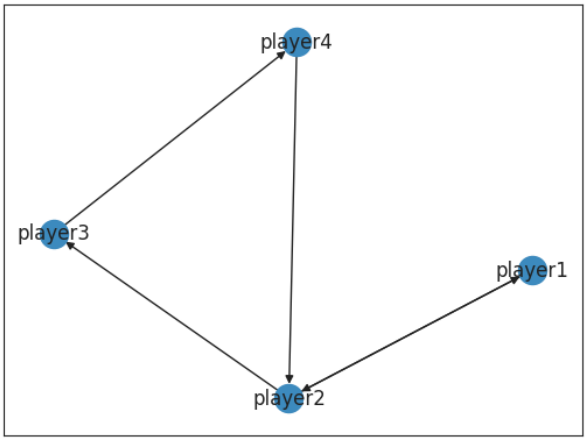
PageRank in pyspark

We will perform PageRank on the WTA matches dataset used before. Before we apply PageRank on the whole dataset, we will try to understand implementation in the pyspark on the small dataset.

match	loser	winner
1	player1	player2
2	player2	player3
3	player3	player4
4	player2	player1
5	player4	player2

```
import networkx as nx
mini_matches = [('player1', 'player2'),
                ('player2', 'player3'),
                ('player3', 'player4'),
                ('player2', 'player1'),
                ('player4', 'player2')]


G = nx.DiGraph()
G.add_edges_from(mini_matches)
nx.draw_networkx(G)
```



First, we are going to create Rdd object from links, and we will group them by source node. For each player we will get the list of matches in which player lost.

```
import pyspark
xs = sc.parallelize(mini_matches)
```

```
links = xs.groupByKey().mapValues(list)
links.collect()
```

```
-----------------------
[('player1', ['player2']),
 ('player2', ['player3', 'player1']),
 ('player3', ['player4']),
 ('player4', ['player2'])]
```

Then we initialize the PageRank od each player.

```
Nodes = (xs.keys() + xs.values()).distinct()
ranks = Nodes.map(lambda x: (x, 100))
sorted(ranks.collect())
-------------------------
[('player1', 1.0), ('player2', 1.0), ('player3', 1.0), ('player4', 1.0)]
```

Here we join lists of lost games and PageRank into tupple. As links and ranks have same keys, we can use that by calling `join()` function.

```
links.join(ranks).collect()
-----------------------------------
links.join(ranks).collect()

[('player2', (['player3', 'player1'], 1.0)),
 ('player3', (['player4'], 1.0)),
 ('player4', (['player2'], 1.0)),
 ('player1', (['player2'], 1.0))]
```

We are going to compute contributions of each node, and finally we will run several iterations of PageRank algorithm.

```
from operator import add

def computeContribs(node_links_rank):
    _, (links, rank) = node_links_rank
    nb_links = len(links)
    for outnode in links:
        yield outnode, rank / nb_links

for iteration in range(10):
    contribs = links.join(ranks).flatMap(computeContribs)
    contribs = links.fullOuterJoin(contribs).mapValues(lambda x : x[1] or 0.0)
    ranks = contribs.reduceByKey(add)
    ranks = ranks.mapValues(lambda rank: rank * 0.85 + 0.15)

    print(sorted(ranks.collect()))
```

Let us bring everything together into function, where we can controll the dumping factor \(\b\) and the number of iterations. Finally we can plot our mini graph, such that nodes with higher PageRank differ in size and colour.

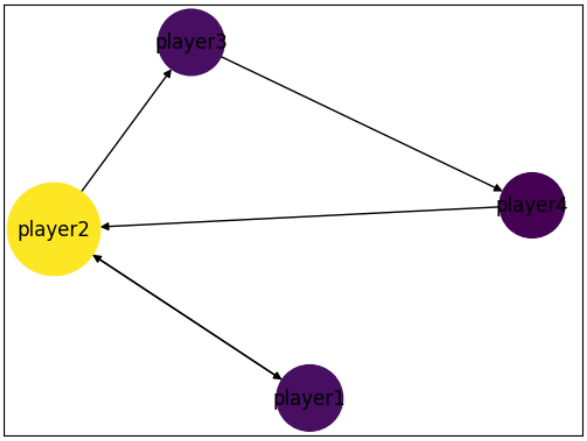
```
def get_page_rank(xs, beta=0.85, niter=10):

    links = xs.groupByKey().mapValues(list)
    Nodes = (xs.keys() + xs.values()).distinct()
    ranks = Nodes.map(lambda x: (x, 100))

    for iteration in range(niter):
        contribs = links.join(ranks).flatMap(computeContribs)
        contribs = links.fullOuterJoin(contribs).mapValues(lambda x : x[1] or 0.0)
        ranks = contribs.reduceByKey(add)
        ranks = ranks.mapValues(lambda rank: rank * beta + (1-beta))

    return ranks
xs = sc.parallelize(mini_matches)
page_ranks = get_page_rank(xs)
r = sorted(page_ranks.collect())

page_rank = [x[1]*100 for x in r]
G = nx.DiGraph()
G.add_edges_from(mini_matches)
nx.draw_networkx(G, node_size=page_rank, node_color=page_rank)
```



PageRank for tennis dataset

Now, we can compute the page rank of any graph. We'll need to preprocess the WTA matches dataset:

```
def get_loser_winner(match):
    ms = match.split(',')
    return (ms[18], ms[10]) #loser, winner

match_data = sc.textFile("tennis_wta-master/wta_matches*")
xs = match_data.map(get_loser_winner) #rdd

page_rank = get_page_rank(xs, beta=0.85, niter=10)

sorted(page_rank.collect(), key=lambda x: x[1], reverse=True)[:10]
```

When we execute it we'll get the most influential tennis players according to PageRank measure:

```
[('Martina Navratilova', 5474.833387989397),
 ('Chris Evert', 4636.296862841747),
 ('Steffi Graf', 3204.060329739364),
 ('Serena Williams', 3039.4891463181716),
 ('Venus Williams', 2737.6910644598042),
 ('Lindsay Davenport', 2544.223902071716),
 ('Billie Jean King', 2258.2918906684013),
 ('Arantxa Sanchez Vicario', 2113.238892322328),
 ('Virginia Wade', 2064.516569589297),
 ('Monica Seles', 2028.8803038982473)]
```

4.3 Machine Learning in PySpark

Apache Spark offers a machine learning API called MLlib, where we can find a different kinds of machine learning algorithms, such as mllib.classification, mllib.linalg, mllib.recommendation, mllib.regression, mllib.clustering. Another usefull module is pyspark.sql which can be used to create **DataFrame** object.

Lets explore K-means clustering with MLlib library. K-means is unsupervised machine learning algorithm that partitions a dataset into K clusters. To demonstrate how K-means works we will use "iris.csv" dataset.

First we need to create SparkSession, and for that we can use:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("mlwithspark").getOrCreate()
```

Then load iris dataset into table:

```
df = spark.read.csv('iris.csv', inferSchema=True, header=True)
df.show()
```

```
|sepal_length|sepal_width|petal_length|petal_width|species|
+------------+-----------+------------+-----------+-------+
|         5.1|        3.5|         1.4|        0.2| setosa|
|         4.9|        3.0|         1.4|        0.2| setosa|
|         4.7|        3.2|         1.3|        0.2| setosa|
|         4.6|        3.1|         1.5|        0.2| setosa|
|         5.0|        3.6|         1.4|        0.2| setosa|
|         5.4|        3.9|         1.7|        0.4| setosa|
```

```
|         4.6|        3.4|         1.4|        0.3| setosa|
|         5.0|        3.4|         1.5|        0.2| setosa|
|         4.4|        2.9|         1.4|        0.2| setosa|
|         4.9|        3.1|         1.5|        0.1| setosa|
|         5.4|        3.7|         1.5|        0.2| setosa|
|         4.8|        3.4|         1.6|        0.2| setosa|
|         4.8|        3.0|         1.4|        0.1| setosa|
|         4.3|        3.0|         1.1|        0.1| setosa|
|         5.8|        4.0|         1.2|        0.2| setosa|
|         5.7|        4.4|         1.5|        0.4| setosa|
|         5.4|        3.9|         1.3|        0.4| setosa|
|         5.1|        3.5|         1.4|        0.3| setosa|
|         5.7|        3.8|         1.7|        0.3| setosa|
|         5.1|        3.8|         1.5|        0.3| setosa|
+------------+-----------+------------+-----------+-------+
only showing top 20 rows
```

Dataset analasys

```
print(df.count(), len(df.columns))
```

```
150 5
```

```
df.columns
```

```
['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'species']
```

```
df.printSchema()
```

```
root
 |-- sepal_length: double (nullable = true)
 |-- sepal_width: double (nullable = true)
 |-- petal_length: double (nullable = true)
 |-- petal_width: double (nullable = true)
 |-- species: string (nullable = true)
```

```
df.select('species').distinct().show()
```

```
+----------+
|   species|
+----------+
| virginica|
|versicolor|
|    setosa|
+----------+
```

```
df.groupBy('species').count().orderBy('count').show()
```

```
+----------+-----+
|   species|count|
+----------+-----+
| virginica|   50|
|versicolor|   50|
|    setosa|   50|
+----------+-----+
```

If we want to add new column to DataFrame:

```
df.withColumn("petal_area", (df['petal_length']*df['petal_width'])).show()
```

```
+------------+-----------+------------+-----------+-------+------------------+
|sepal_length|sepal_width|petal_length|petal_width|species|        petal_area|
+------------+-----------+------------+-----------+-------+------------------+
|         5.1|        3.5|         1.4|        0.2| setosa|0.27999999999999997|
|         4.9|        3.0|         1.4|        0.2| setosa|0.27999999999999997|
|         4.7|        3.2|         1.3|        0.2| setosa|              0.26|
|         4.6|        3.1|         1.5|        0.2| setosa|0.30000000000000004|
|         5.0|        3.6|         1.4|        0.2| setosa|0.27999999999999997|
|         5.4|        3.9|         1.7|        0.4| setosa|              0.68|
|         4.6|        3.4|         1.4|        0.3| setosa|              0.42|
|         5.0|        3.4|         1.5|        0.2| setosa|0.30000000000000004|
|         4.4|        2.9|         1.4|        0.2| setosa|0.27999999999999997|
|         4.9|        3.1|         1.5|        0.1| setosa|0.15000000000000002|
|         5.4|        3.7|         1.5|        0.2| setosa|0.30000000000000004|
|         4.8|        3.4|         1.6|        0.2| setosa|0.32000000000000006|
|         4.8|        3.0|         1.4|        0.1| setosa|0.13999999999999999|
|         4.3|        3.0|         1.1|        0.1| setosa|0.11000000000000001|
|         5.8|        4.0|         1.2|        0.2| setosa|              0.24|
|         5.7|        4.4|         1.5|        0.4| setosa| 0.6000000000000001|
|         5.4|        3.9|         1.3|        0.4| setosa|              0.52|
|         5.1|        3.5|         1.4|        0.3| setosa|              0.42|
|         5.7|        3.8|         1.7|        0.3| setosa|              0.51|
```

```
|         5.1|        3.8|         1.5|        0.3| setosa|0.44999999999999996|
+------------+-----------+------------+-----------+-------+-------------------+
only showing top 20 rows
```

```
df.filter(df['species']=='setosa').show()
```

```
+------------+-----------+------------+-----------+-------+
|sepal_length|sepal_width|petal_length|petal_width|species|
+------------+-----------+------------+-----------+-------+
|         5.1|        3.5|         1.4|        0.2| setosa|
|         4.9|        3.0|         1.4|        0.2| setosa|
|         4.7|        3.2|         1.3|        0.2| setosa|
|         4.6|        3.1|         1.5|        0.2| setosa|
|         5.0|        3.6|         1.4|        0.2| setosa|
|         5.4|        3.9|         1.7|        0.4| setosa|
|         4.6|        3.4|         1.4|        0.3| setosa|
|         5.0|        3.4|         1.5|        0.2| setosa|
|         4.4|        2.9|         1.4|        0.2| setosa|
|         4.9|        3.1|         1.5|        0.1| setosa|
|         5.4|        3.7|         1.5|        0.2| setosa|
|         4.8|        3.4|         1.6|        0.2| setosa|
|         4.8|        3.0|         1.4|        0.1| setosa|
|         4.3|        3.0|         1.1|        0.1| setosa|
|         5.8|        4.0|         1.2|        0.2| setosa|
|         5.7|        4.4|         1.5|        0.4| setosa|
|         5.4|        3.9|         1.3|        0.4| setosa|
|         5.1|        3.5|         1.4|        0.3| setosa|
|         5.7|        3.8|         1.7|        0.3| setosa|
|         5.1|        3.8|         1.5|        0.3| setosa|
+------------+-----------+------------+-----------+-------+
only showing top 20 rows
```

```
df.groupBy('species').sum().show()
```

```
+------------+-----------+------------+-----------+-------+
|sepal_length|sepal_width|petal_length|petal_width|species|
+------------+-----------+------------+-----------+-------+
|         5.1|        3.5|         1.4|        0.2| setosa|
|         4.9|        3.0|         1.4|        0.2| setosa|
|         4.7|        3.2|         1.3|        0.2| setosa|
|         4.6|        3.1|         1.5|        0.2| setosa|
|         5.0|        3.6|         1.4|        0.2| setosa|
|         5.4|        3.9|         1.7|        0.4| setosa|
|         4.6|        3.4|         1.4|        0.3| setosa|
|         5.0|        3.4|         1.5|        0.2| setosa|
|         4.4|        2.9|         1.4|        0.2| setosa|
|         4.9|        3.1|         1.5|        0.1| setosa|
|         5.4|        3.7|         1.5|        0.2| setosa|
|         4.8|        3.4|         1.6|        0.2| setosa|
|         4.8|        3.0|         1.4|        0.1| setosa|
|         4.3|        3.0|         1.1|        0.1| setosa|
|         5.8|        4.0|         1.2|        0.2| setosa|
|         5.7|        4.4|         1.5|        0.4| setosa|
|         5.4|        3.9|         1.3|        0.4| setosa|
|         5.1|        3.5|         1.4|        0.3| setosa|
|         5.7|        3.8|         1.7|        0.3| setosa|
|         5.1|        3.8|         1.5|        0.3| setosa|
+------------+-----------+------------+-----------+-------+
only showing top 20 rows
```

```
df.groupBy('species').max().show()
```

```
+----------+-----------------+----------------+-----------------+----------------+
|   species|max(sepal_length)|max(sepal_width)|max(petal_length)|max(petal_width)|
+----------+-----------------+----------------+-----------------+----------------+
| virginica|              7.9|             3.8|              6.9|             2.5|
|versicolor|              7.0|             3.4|              5.1|             1.8|
|    setosa|              5.8|             4.4|              1.9|             0.6|
+----------+-----------------+----------------+-----------------+----------------+
```

Clustering

Before running the K-means algorithm, all features are merged into single column. Then we need to determine the optimal number of clusters (K). We can use elbow method, ploting the error over different values of K, and finding the elbow point.

```
from pyspark.ml.feature import StringIndexer
feature = StringIndexer(inputCol="species", outputCol="targetlabel")
target = feature.fit(df).transform(df)
target.show()
```

```
+------------+-----------+------------+-----------+-------+-----------+
|sepal_length|sepal_width|petal_length|petal_width|species|targetlabel|
+------------+-----------+------------+-----------+-------+-----------+
|         5.1|        3.5|         1.4|        0.2| setosa|        0.0|
```

```
|         4.9|        3.0|         1.4|        0.2| setosa|        0.0|
|         4.7|        3.2|         1.3|        0.2| setosa|        0.0|
|         4.6|        3.1|         1.5|        0.2| setosa|        0.0|
|         5.0|        3.6|         1.4|        0.2| setosa|        0.0|
|         5.4|        3.9|         1.7|        0.4| setosa|        0.0|
|         4.6|        3.4|         1.4|        0.3| setosa|        0.0|
|         5.0|        3.4|         1.5|        0.2| setosa|        0.0|
|         4.4|        2.9|         1.4|        0.2| setosa|        0.0|
|         4.9|        3.1|         1.5|        0.1| setosa|        0.0|
|         5.4|        3.7|         1.5|        0.2| setosa|        0.0|
|         4.8|        3.4|         1.6|        0.2| setosa|        0.0|
|         4.8|        3.0|         1.4|        0.1| setosa|        0.0|
|         4.3|        3.0|         1.1|        0.1| setosa|        0.0|
|         5.8|        4.0|         1.2|        0.2| setosa|        0.0|
|         5.7|        4.4|         1.5|        0.4| setosa|        0.0|
|         5.4|        3.9|         1.3|        0.4| setosa|        0.0|
|         5.1|        3.5|         1.4|        0.3| setosa|        0.0|
|         5.7|        3.8|         1.7|        0.3| setosa|        0.0|
|         5.1|        3.8|         1.5|        0.3| setosa|        0.0|
+------------+-----------+------------+-----------+-------+-----------+
only showing top 20 rows
```

```
input_cols=['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
from pyspark.ml.feature import VectorAssembler

vec_assembler = VectorAssembler(inputCols=input_cols, outputCol='features')
final_data = vec_assembler.transform(target)
final_data.show()
```

```
+------------+-----------+------------+-----------+-------+-----------+-----------------+
|sepal_length|sepal_width|petal_length|petal_width|species|targetlabel|         features|
+------------+-----------+------------+-----------+-------+-----------+-----------------+
|         5.1|        3.5|         1.4|        0.2| setosa|        0.0|[5.1,3.5,1.4,0.2]|
|         4.9|        3.0|         1.4|        0.2| setosa|        0.0|[4.9,3.0,1.4,0.2]|
|         4.7|        3.2|         1.3|        0.2| setosa|        0.0|[4.7,3.2,1.3,0.2]|
|         4.6|        3.1|         1.5|        0.2| setosa|        0.0|[4.6,3.1,1.5,0.2]|
|         5.0|        3.6|         1.4|        0.2| setosa|        0.0|[5.0,3.6,1.4,0.2]|
|         5.4|        3.9|         1.7|        0.4| setosa|        0.0|[5.4,3.9,1.7,0.4]|
|         4.6|        3.4|         1.4|        0.3| setosa|        0.0|[4.6,3.4,1.4,0.3]|
|         5.0|        3.4|         1.5|        0.2| setosa|        0.0|[5.0,3.4,1.5,0.2]|
|         4.4|        2.9|         1.4|        0.2| setosa|        0.0|[4.4,2.9,1.4,0.2]|
|         4.9|        3.1|         1.5|        0.1| setosa|        0.0|[4.9,3.1,1.5,0.1]|
|         5.4|        3.7|         1.5|        0.2| setosa|        0.0|[5.4,3.7,1.5,0.2]|
|         4.8|        3.4|         1.6|        0.2| setosa|        0.0|[4.8,3.4,1.6,0.2]|
|         4.8|        3.0|         1.4|        0.1| setosa|        0.0|[4.8,3.0,1.4,0.1]|
|         4.3|        3.0|         1.1|        0.1| setosa|        0.0|[4.3,3.0,1.1,0.1]|
|         5.8|        4.0|         1.2|        0.2| setosa|        0.0|[5.8,4.0,1.2,0.2]|
|         5.7|        4.4|         1.5|        0.4| setosa|        0.0|[5.7,4.4,1.5,0.4]|
|         5.4|        3.9|         1.3|        0.4| setosa|        0.0|[5.4,3.9,1.3,0.4]|
|         5.1|        3.5|         1.4|        0.3| setosa|        0.0|[5.1,3.5,1.4,0.3]|
|         5.7|        3.8|         1.7|        0.3| setosa|        0.0|[5.7,3.8,1.7,0.3]|
|         5.1|        3.8|         1.5|        0.3| setosa|        0.0|[5.1,3.8,1.5,0.3]|
+------------+-----------+------------+-----------+-------+-----------+-----------------+
only showing top 20 rows
```

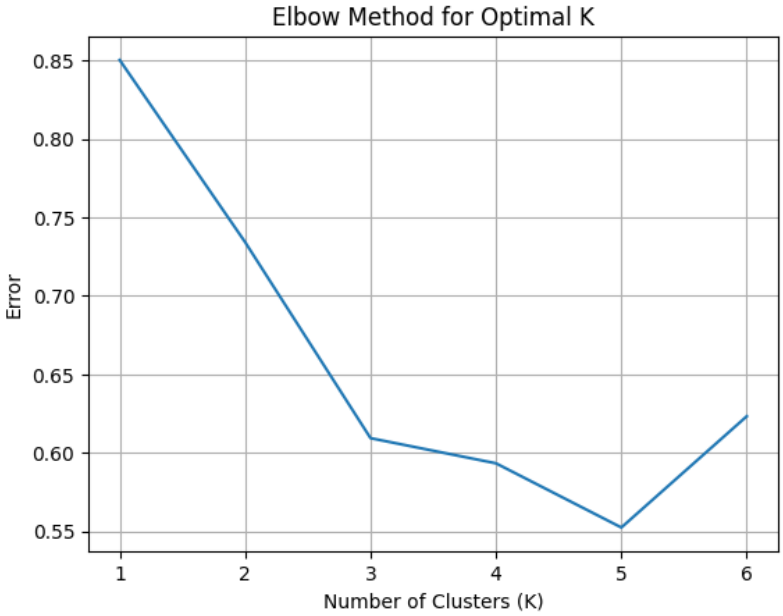
```
from pyspark.ml.clustering import KMeans
from pyspark.ml.evaluation import ClusteringEvaluator
import matplotlib.pyplot as plt

# Computing WSSSE for K values from 2 to 8
errors =[]
evaluator = ClusteringEvaluator(predictionCol='prediction', featuresCol='features', \
                                metricName='silhouette', distanceMeasure='squaredEuclidean')

for i in range(2,8):
    KMeans_mod = KMeans(featuresCol='features', k=i)
    KMeans_fit = KMeans_mod.fit(final_data)
    output = KMeans_fit.transform(final_data)
    score = evaluator.evaluate(output)
    errors.append(score)
    print("K=%s, error=%s"%(i, score))
```

```
K=2, error=0.8501515983265806
K=3, error=0.7342113066202725
K=4, error=0.6093888373825749
K=5, error=0.5933815144059972
K=6, error=0.5524969270291149
K=7, error=0.6233281829975698
```

```
plt.plot(range(1, 7), errors)
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Error')
plt.title('Elbow Method for Optimal K')
plt.grid()
plt.show()
```

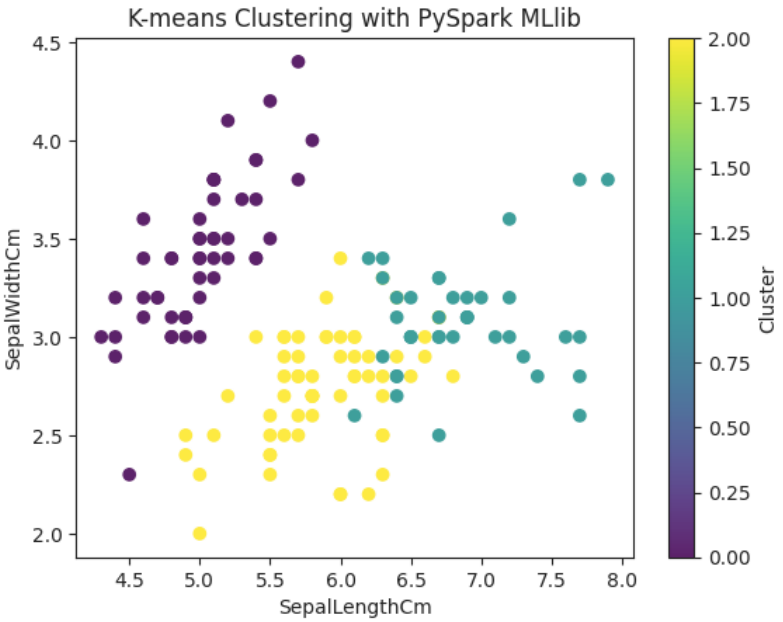


Finally, we can run kmeans for 3 clusters and visualize the resulting clusters.

```
kmeans = KMeans(k=3, featuresCol="features", predictionCol="cluster")
kmeans_model = kmeans.fit(final_data)
clustered_data = kmeans_model.transform(final_data)
```

```
import pandas as pd
# Converting to Pandas DataFrame
clustered_data_pd = clustered_data.toPandas()

# Visualizing the results
plt.scatter(clustered_data_pd["sepal_length"], clustered_data_pd["sepal_width"], c=clustered_data_pd["cluster"], cmap='viridis')
plt.xlabel("SepalLengthCm")
plt.ylabel("SepalWidthCm")
plt.title("K-means Clustering with PySpark MLlib")
plt.colorbar().set_label("Cluster")
plt.show()
```



5. Used Datasets and packages

5.1 Datasets

- Moby Dick book, we will get file `pg2701.txt`

```
wget "https://gutenberg.org/cache/epub/2701/pg2701.txt"
```

- Tennis WTA matches can be downloaded from the github repository

```
wget "https://github.com/JeffSackmann/tennis_wta/archive/refs/heads/master.zip"
unzip master.zip
```

The dataset contains `.csv files` with WTA matches from 1968 until 2023.

- Iris Flowers Dataset can be downloaded from many sources, in this tutorial I used one from Kaggle

5.2 Requirement

- python3
- mrjob `pip install mrjob`
- pyspark `pip install pyspark` To use pyspark you need previosly installed java.
- networkx `pip install networkx`.
- matplotlib `pip install matplotlib`
- pandas `pip install pandas`

5.3 Literature

- MapReduce: Simplified Data Processing on Large Clusters
- Page Rank paper, The PageRank Citation Ranking: Bringig Order to the Web
- Wolohan, J. (2020). Mastering Large Datasets with Python: Parallelize and Distribute Your Python Code. United States: Manning.
- Radtka, Z., & Miner, D. (2015). Hadoop with Python. O'Reilly Media.
- Tutorial BigData with PySpark by New York University
- CornellEdu BigData Technologies course
- Paradox Hadoop user guide