Initialisation des imports :

from random import sample, random

Initialisation du jeu de données

# Initialize data

dat\_1=sc.parallelize(range(1, 5+1))

coll = dat\_1.collect()

print ("Elements in RDD -> ",coll)

dat\_2=sc.parallelize(range(1, 5+1))

coll = dat\_2.collect()

print ("Elements in RDD -> ",coll)

words = sc.parallelize (["scala", "java", "hadoop", "spark", "akka","spark vs hadoop","pyspark", "pyspark and spark"])

coll = words.collect()

print ("Elements in RDD -> %s" % coll)

**Transformation**

exercice 1  utilisation des maps:

construire des tuples avec les fonctions maps

words\_map = words.map(lambda x: (x, 1))

mapping = words\_map.collect()

print "Key value pair -> %s" % (mapping)

dat\_1=dat\_1.map(lambda x:(x, sample(["a", "b","c"], 1)[0]))

dat\_2=dat\_2.map(lambda x:(x, random()))

dat\_2.persist() # we need to cache dat\_2 otherwise it will generate random every time dat\_2 is invoked

exercice 2 : utilisation des flatMap

dat\_1.flatMap(lambda x:x).collect()

exercice 3 : utilisation des filter

dat\_1.filter(lambda x:x[0] in [1,2,3]).collect()

exercice 4 : utilisation des union

dat=dat\_1.union(dat\_2)

dat.collect()

exercice 5 : utilisation des intersections

dat\_1.intersection(dat\_1).collect()

dat\_1.intersection(dat\_2).collect()

exercice 6: utilisation des distinsct

dat\_1.union(dat\_1).collect()

dat\_1.union(dat\_1).distinct().collect()

exercice 7 : utilisation des groupByKey

dat\_1.groupByKey().collect()

dat\_1.groupByKey().collect()[0]

dat\_1.groupByKey().collect()[0][1]

dat\_1.groupByKey().collect()[0][1].data

exercice 8 : utilisation des reduceByKey

# "reduceByKey" operation

# function must be of type (V,V) => V

dat\_2.collect()

# Here I union dat\_2 with itself and then reduce by key to get the sum of all values of each key. So the results we get below should be the double of the values above

dat\_2.union(dat\_2).reduceByKey(lambda x,y:x+y).collect()

exercice 9 : utilisation des aggregateByKey

# "aggregateByKey" operation

dat\_2.union(dat\_2).aggregateByKey(100, lambda a,b:a+b, lambda x,y:x+y).collect()

dat\_2.union(dat\_2).aggregateByKey(100, # the initial value. will act as "a" below

lambda a,b:a+b, # "b" is the value from RDD

lambda x,y:x+y).collect() # function to reduce

# a more practical example of "aggregateByKey"

# here we generate many random numbers for keys 1,2,3,4,

# then calculate the average value for each key

dat\_3=sc.parallelize(list("1223334444")\*1000)

dat\_3=dat\_3.map(lambda x:(x, random()))

dat\_3=dat\_3.aggregateByKey((0.0,0), lambda a,b:(a[0]+b, a[1]+1),

lambda rdd1,rdd2:(rdd1[0]+rdd2[0], rdd1[1]+rdd2[1]))

dat\_3.collect()

dat\_3.mapValues(lambda x:x[0]/x[1]).collect()

# or

dat\_3.map(lambda x:(x[0], x[1][0]/x[1][1])).collect()

exercice 10 : utilisation des aggregate

# "aggregate" operation

sc.parallelize([1,2,3,4]).aggregate(

(0, 0.0),

lambda acc, value: (acc[0] + 1, acc[1] + value),

lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1]))

exercice 11 : utilisation des sortByKey

# "sortByKey" operation

dat\_2.union(dat\_2).collect()

dat\_2.union(dat\_2).sortByKey().collect()

exercice 12 : utilisation des jointures

dat\_1.join(dat\_2).collect()

#jointure

x = sc.parallelize([("spark", 1), ("hadoop", 4)])

y = sc.parallelize([("spark", 2), ("hadoop", 5)])

joined = x.join(y)

final = joined.collect()

print "Join RDD -> %s" % (final)

exercice 13 : utilisation des cartesian

# "cartesian" operation

dat\_1.count()

dat\_2.count()

dat\_1.cartesian(dat\_2).count()

dat\_1.cartesian(dat\_2).collect()

**Actions**

exercice 1 : utilisation des reduce

# "reduce" operation

sc.parallelize(range(1, 101)).reduce(lambda a,b:a+b)

exercice 2 : utilisation des count

# "count" operation

sc.parallelize(range(100)).count()

exercice 3 : utilisation des first

# "first" operation

sc.parallelize([5,4,3]).first()

exercice 4 : utilisation des take

# "take" operation

sc.parallelize([5,4,3,2,1]).take(2)

exercice 5 : utilisation des takeSample

# "takeSample" operation

# 1st argument is "if with replacement". 1 is true, 0 is false

# 2nd argument is how many observations to sample

# 3rd argument, random seed, is optional

sc.parallelize([5,4,3,2,1]).takeSample(1, 6)

exercice 6 : utilisation des takeOrdered

# "takeOrdered" operation

# the arugment is the number of elements to take.

sc.parallelize([10,4,5,3,2]).takeOrdered(3)

exercice 7 : utilisation des countByKey

# "countByKey" Operation

sc.parallelize(list("1223334444")\*1000).countByKey()

exercice 8 : utilisation des foreach

# "foreach" and accumulator operation

accum = sc.accumulator(0)

sc.parallelize(range(1, 100+1)).foreach(lambda x: accum.add(x))

accum.value

exercice 9 : utilisation des accumulator

#Accumulator variables are used for aggregating the information through associative and commutative operations. For example, you can use an accumulator for a sum operation or counters (in MapReduce).

Exemple 1 :

num = sc.accumulator(15)

def f(x):

global num

num+=x

rdd = sc.parallelize([20,30,40,50])

rdd.foreach(f)

final = num.value

print "Accumulated value is -> %i" % (final)

Exemple 2 :

accum = sc.accumulator(0)

sc.parallelize(range(1, 100+1)).foreach(lambda x: accum.add(1))

accum.value

# NOTE: accumulator can only be updated during actions, and will not happen during transformations

# hence maybe better to use accumulator together with "foreach" as "foreach" itself is ACTION rather than TRANSFORMATION

accum = sc.accumulator(0)

sc.parallelize(range(100)).map(test)

accum.value # 0

sc.parallelize(range(100)).map(test).collect()

accum.value # 100

spark SQL :

exercice UDF SQL :

from pyspark.sql.types import StringType

from pyspark.sql.functions import udf

maturity\_udf = udf(lambda age: "adult" if age >=18 else "child", StringType())

df = sqlContext.createDataFrame([{'name': 'Alice', 'age': 1}])

df.withColumn("maturity", maturity\_udf(df.age))