Reduce and Fold in Spark

We have two commonly used RDD functions reduce and fold in Spark, and this article mainly talks about their similarity and difference, and under what scenarios should they be used.

Looking at the signatures of two methods, we find that both reduce and fold accept a function, but fold in addition take a zeroValue to be used for the initiali call of each partition.

def reduce(self, op):

...

def fold(self, zeroValue, op):

Similarity:

Most of the time, we do not really get bothered by the difference between the two.

rdd = sc.parallelize([2, 3, 4])

# reduction by addition

rdd.reduce(lambda x, y: x+y)

# 9

rdd.fold(0, lambda x, y: x+y)

# 9

# reduction by multiplication

rdd.reduce(lambda x, y: x\*y)

# 24

rdd.fold(1, lambda x, y: x\*y)

# 24

The extra parameter required in reduce is zeroValue. However, zeroValue has different values in different scenarios. In general, it is an identify element for your operation. That is, for addition, any number can be added by 0 and remains the same; for multiplication, any number can be multiplied by 1 and remains the same.

As long as we are careful with the choice of zeroValue, reduce and fold can be used interchangeably.

Hmm, what if we are not careful with zeroValue?

Difference

For addition zeroValue is 0. What if we make a mistake and choose 1 for addition? Let’s see an example.

rdd = sc.parallelize([2, 3, 4])

# reduction by addition

rdd.reduce(lambda x, y: x+y)

# 9

rdd.fold(1, lambda x, y: x+y)

# 14

Note that the results by reduce still remain the same while the results by fold is obviously wrong. But where does it get wrong? 2 + 3 + 4 = 9 from reduce, and how does 2 + 3 + 4 + ? = 14 from fold? Let’s first see how many partitions there are for the RDD.

rdd = sc.parallelize([2, 3, 4])

rdd.getNumPartitions()

# 4

rdd.glom().collect()

# [[], [2], [3], [4]]

Now that we know there are 4 partitions in the RDD, let’s slip in some print statements to the operation that we pass to RDD function, to get a glimpse of what happens at each step.

rdd = sc.parallelize([2, 3, 4])

def add(x, y):

print '{}+{}={}'.format(x, y, x+y)

return x+y

rdd.reduce(add)

# 2+3=5

# 5+4=9

# 9

rdd.fold(1, add)

# 1+3=4

# 1+2=3

# 1+4=5

# 1+1=2

# 2+3=5

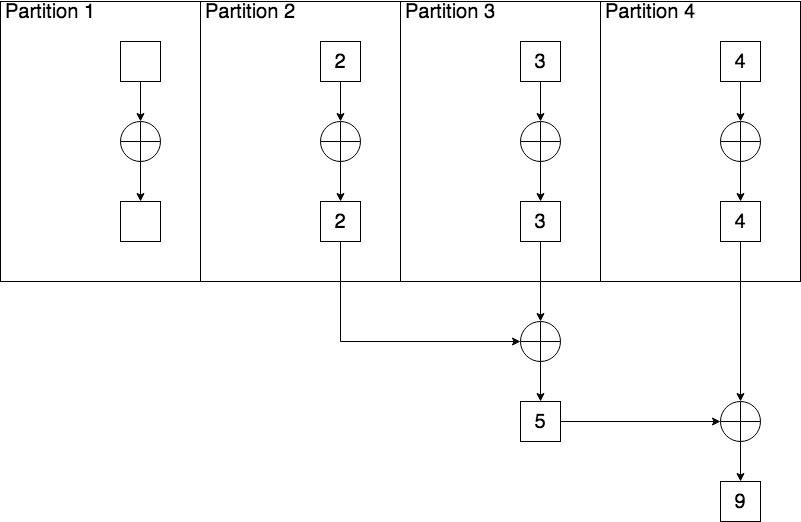
# 5+4=9

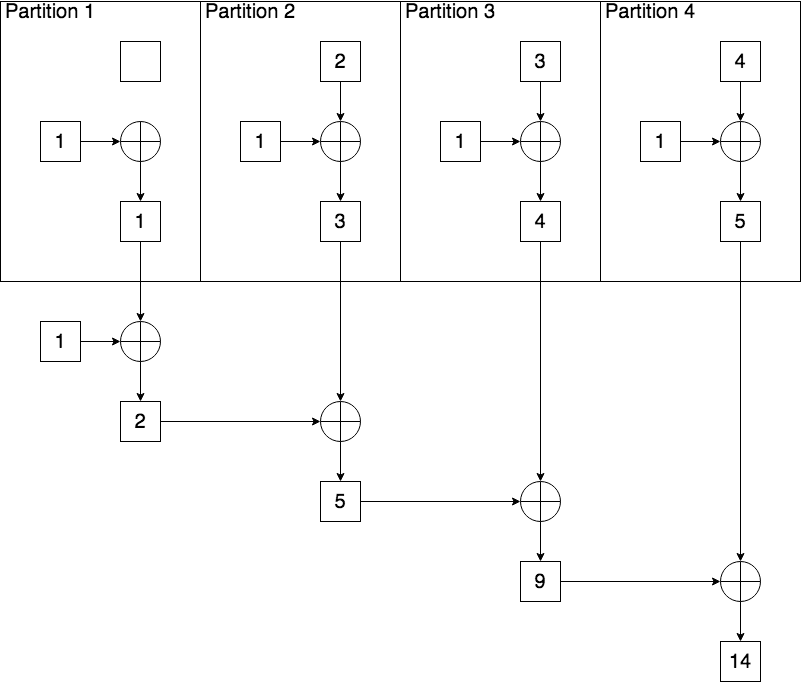
# 9+5=14

# 14

They may seen a bit messy at the first sight. Let’s illustrate with two flowcharts.

Reduce





Empty RDD

If you run reduce on an empty RDD, you will come across the following error

rdd = sc.parallelize([])

rdd.reduce(lambda x, y: x+y)

# Traceback (most recent call last):

# File "<stdin>", line 1, in <module>

# File "/Users/yuanxu/spark-2.2.0-bin-hadoop2.7/python/pyspark/rdd.py", line 838, in reduce

# raise ValueError("Can not reduce() empty RDD")

# ValueError: Can not reduce() empty RDD

But you can safely run fold on it with a correct zeroValue

rdd = sc.parallelize([])

rdd.fold(0, lambda x, y: x+y)

# 0

In this case, zeroValue is playing the role of a default value for an empty RDD.

Bar raiser

If you are running an auction for your old iPhone 6 on eBay, you would like to sell it to the buyer with the highest bid.

However, you also have a threshold under which you would rather keep it yourself. Let’s say, your threshold is $300.

You will finally sell your lovely iPhone 6 to the buyer whose bid price is the highest and also larger than $300, or

set the price to $300 if no one bids more than $300.

rdd = sc.parallelize([280, 250, 210, 285, 100])

rdd.fold(300, lambda x, y: max(x, y))

# 300

You run the program above, and sadly find that no one bids more than $300.

You finally set the price to your threshold which is $300 and hope someone would take it someday.