SPARK ACTIONS IN DEPTH

By www.HadoopExam.com

Note: These instructions should be used with the HadoopExam Apache Spark: Professional Trainings. Where it is executed and you can do hands on with trainer.

Apache Spark Action in Depth

⇒ Action will force the evaluation of the transformation (Lazy)

⇒ Common Actions are

reduce():- Which takes a function argument, which works on two elements.

+ ⇒ is a function, which work on two elements

```
a+b
```

List(1,2,3,4).reduce(_+_) ⇒ Will produce 10

((1+2)+3)+4

6

10

reduce function iteratively called on each element.

```
Val sum = rdd.reduce()
```

fold():- its similar to reduce

- reduce and fold works same way
- Both has same performance
- Difference:- it takes as zero value as initial value

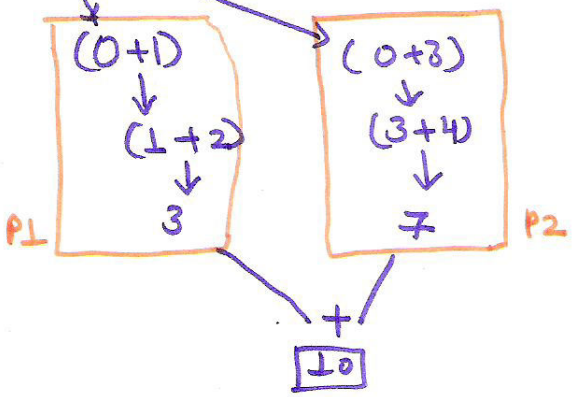
int → 0 [Summation] +

int → 1 [Multiplication] *

Empty collection → for concatenation

P1 P2

List(1,2,3,4).fold(0)(_ + _) ⇒ Produce 10



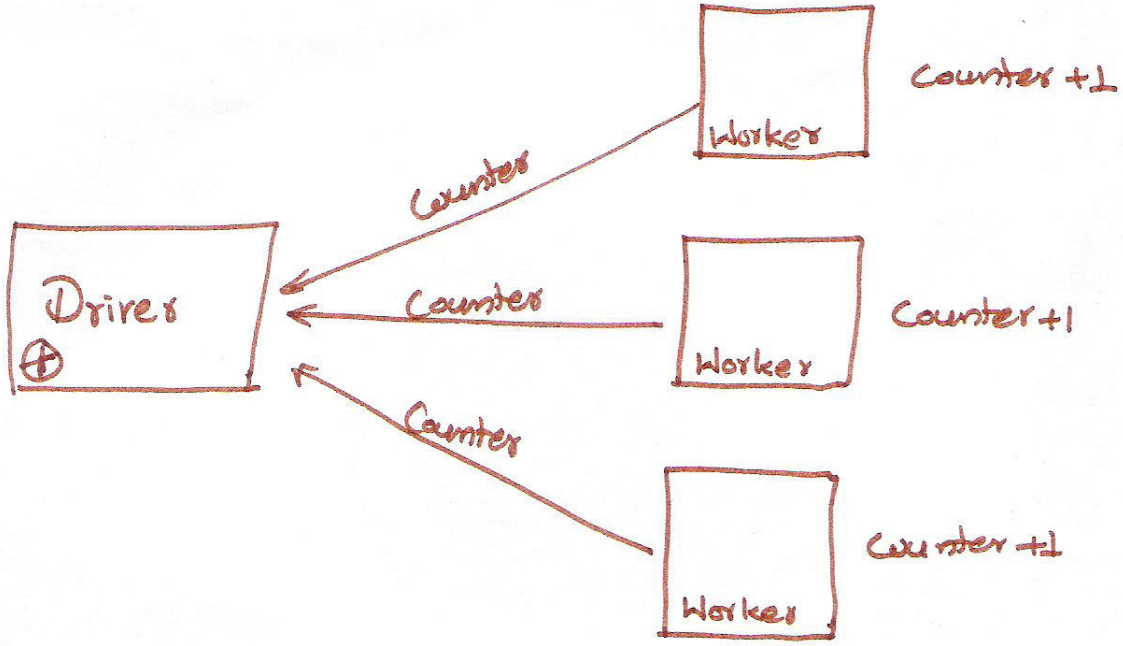
⇒ Reduce & fold, both have same return type as input rdd

aggregate() :- It would have different return type.

- In aggregate, also we will have initial zero value of type we want to return.
- It requires ~~two~~ two function as input

Accumulator :- Let's first understand Accumulator

⇒ Accumulators are similar to counter in Hadoop framework.



⇒ Accumulators are updated by worker node only (write only)

⇒ At worker node, you can not read Accumulator value.

⇒ Accumulator values can be read only at Driver program

⇒ Example:-

During the Job execution, you can count empty/bad lines.

Example

```
Val accum = sc.accumulator(0, "name")
sc.parallelize(Array(1,2,3,4)).foreach
                (x ⇒ accum+x)
accum.value
  =10
```

=> Accumulator are only added through an associative operation.

=> They are efficiently supported in parallel.

=> If accumulators are created with a name, they will be displayed in Spark's UI.

=> Similar to counter it can be useful for understanding the progress of running stages.

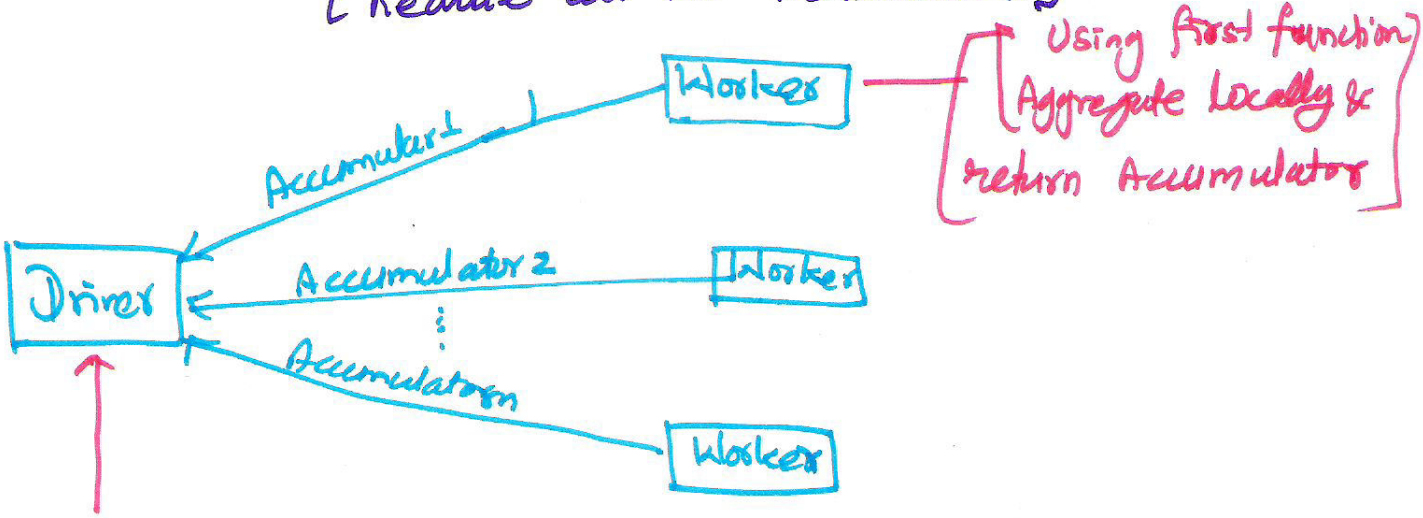
=> Accumulators do not change the lazy evaluation model of Spark.

Aggregate() again:- It requires two function.

→ first function:- To combine values locally on each partitions using Accumulator.

→ Second function:- merge two Accumulators

[Reduce all the Accumulators]



Drivers will merge all the Accumulator using second function.

⇒ If we want to get average of running number

[1,2,3,4,5]

↓

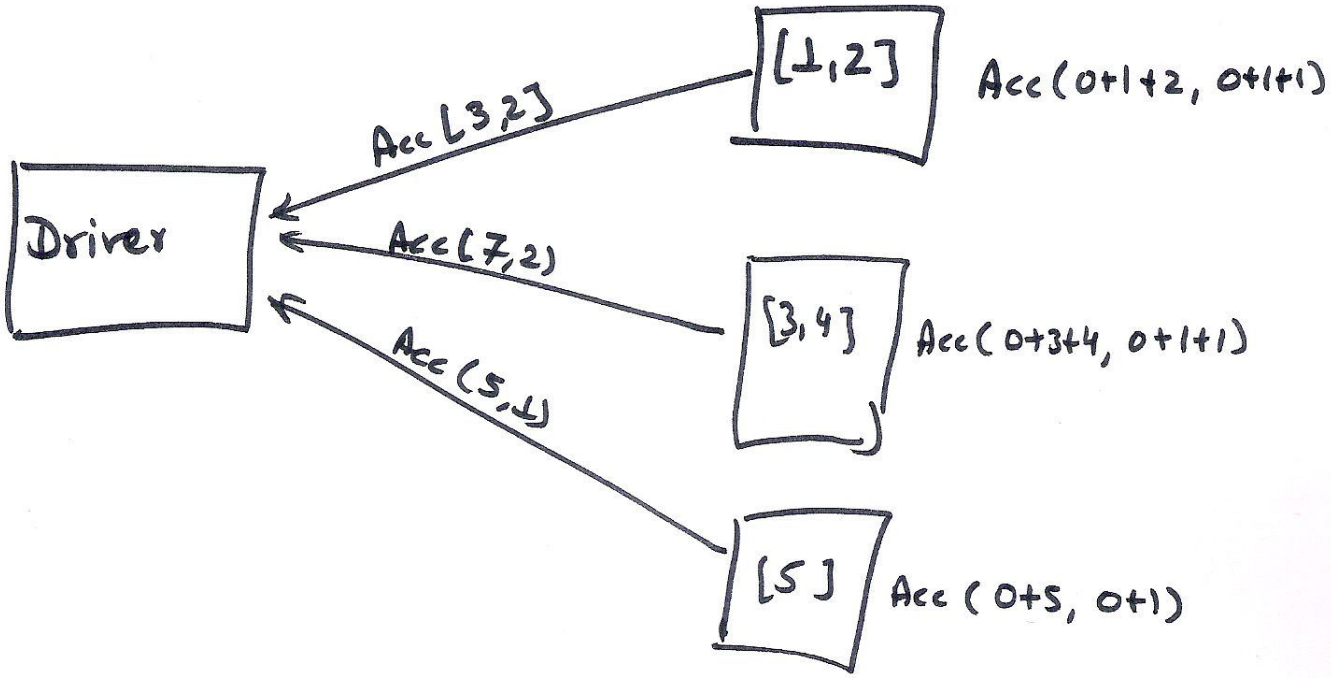
input.aggregate(0,0)(

(acc, value) ⇒ (acc._1 + value, acc._2 + 1)

)

~~Average~~ =

⇒ Accumulator is holding a tuple here, with initial value as zero



Second function

(acc1, acc2) ⇒ (acc._1 + acc2._1, acc1._2 + acc2._2)

⇓

result(15, 5)

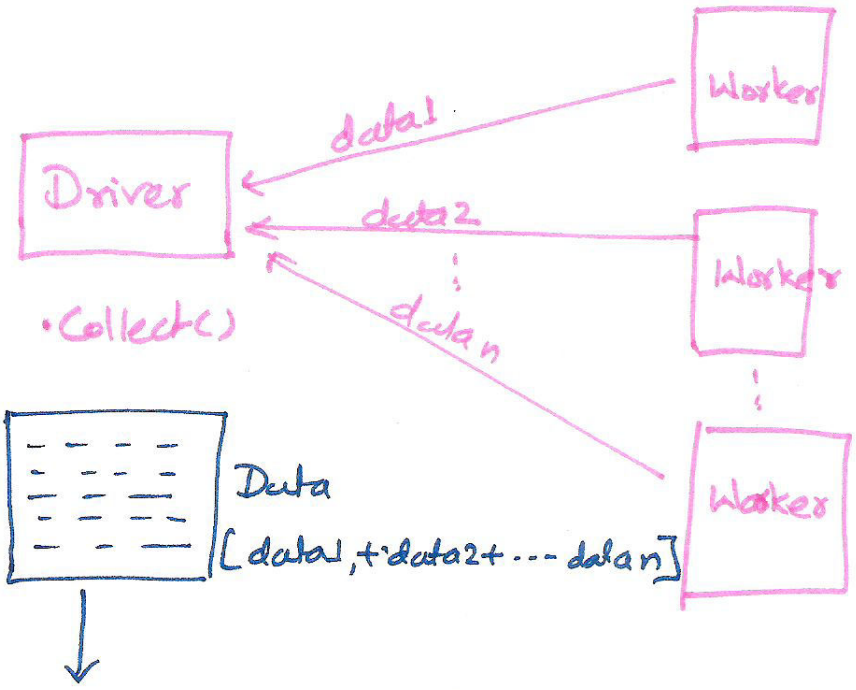
Calculate the Average = result._1 / result._2.toDouble

= 3.0

Action which return Collections to data to Driver

collect() :-

- It returns the entire RDD's contents to Driver



[This entire data must fit in memory of Driver program].

⇒ take(n) :- Returns the n elements from the RDD

- If RDD is partitioned, it minimizes number of partition it access.
- It's order is not fixed.

⇒ Ordered Access :-

top() :- It will be using default ordering on data

- We can also supply our own comparision function to extract top elements

⇒ ~~foreach~~ :-

⇒ foreach()

- perform an action on all elements in RDD
- It works on each worker node independently.
- foreach, will not return data to driver program.

⇒ count(): Returns number of elements in the RDD.

⇒ countByValue(): Number of times each element occurs in RDD

```
rdd.countByValue()
```

- **reduce**: Please note that any function *f* you provide, should be commutative in order to generate reproducible results.

  def reduce(f: (T, T) => T): T

```
val a = sc.parallelize(1 to 100, 3)
a.reduce(_ + _) //res41: Int = 5050
```

- **fold** : Aggregates the values of each partition. The aggregation variable within each partition is initialized with *zeroValue*.

  def fold(zeroValue: T)(op: (T, T) => T): T

```
val a = sc.parallelize(List(1,2,3), 3)
a.fold(0)(_ + _)  //res59: Int = 6
```

- **aggregate** : The aggregate function allows the user to apply two different reduce functions to the RDD. The first reduce function is applied within each partition to reduce the data within each partition into a single result. The second reduce function is used to combine the different reduced results of all partitions together to arrive at one final result. The ability to have two separate reduce functions for intra partition versus across partition reducing adds a lot of flexibility. For example the first reduce function can be the max function and the second one can be the sum function. The user also specifies an initial value. Here are some important facts.

- The initial value is applied at both levels of reduce. So both at the intra partition reduction and across partition reduction.
- Both reduce functions have to be commutative and associative.
- Do not assume any execution order for either partition computations or combining partitions.

  def aggregate[U: ClassTag](zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U): U

```
val z = sc.parallelize(List(1,2,3,4,5,6), 2)

// lets first print out the contents of the RDD with partition labels
def myfunc(index: Int, iter: Iterator[(Int)]) : Iterator[String] = {
  iter.toList.map(x => "[partID:" +  index + ", val: " + x + "]").iterator
}

z.mapPartitionsWithIndex(myfunc).collect
z.aggregate(0)(math.max(_, _), _ + _)
res40: Int = 9
```

```
// This example returns 16 since the initial value is 5
```

```
// reduce of partition 0 will be max(5, 1, 2, 3) = 5
// reduce of partition 1 will be max(5, 4, 5, 6) = 6
// final reduce across partitions will be 5 + 5 + 6 = 16
// note the final reduce include the initial value
z.aggregate(5)(math.max(_, _), _ + _)
```

```
val z = sc.parallelize(List("a","b","c","d","e","f"),2)

//lets first print out the contents of the RDD with partition labels
def myfunc(index: Int, iter: Iterator[(String)]) : Iterator[String] = {
  iter.toList.map(x => "[partID:" +  index + ", val: " + x + "]").iterator
}

z.mapPartitionsWithIndex(myfunc).collect
z.aggregate("")(_ + _, _+_)

// See here how the initial value "x" is applied three times.
//  - once for each partition
//  - once when combining all the partitions in the second reduce function.
z.aggregate("x")(_ + _, _+_)
```

```
val z = sc.parallelize(List("12","23","345","4567"),2)
z.aggregate("")((x,y) => math.max(x.length, y.length).toString, (x,y) => x + y) //res141: String = 42

z.aggregate("")((x,y) => math.min(x.length, y.length).toString, (x,y) => x + y) //res142: String = 11

val z = sc.parallelize(List("12","23","345",""),2)
z.aggregate("")((x,y) => math.min(x.length, y.length).toString, (x,y) => x + y) //res143: String = 10
```

- countByValue : Returns a map that contains all unique values of the RDD and their respective occurrence counts. *(Warning: This operation will finally aggregate the information in a single reducer.)*

  def countByValue(): Map[T, Long]

```
val b = sc.parallelize(List(1,2,3,4,5,6,7,8,2,4,2,1,1,1,1,1))
b.countByValue
res27: scala.collection.Map[Int,Long] = Map(5 -> 1, 8 -> 1, 3 -> 1, 6 -> 1, 1 -> 6, 2 -> 3, 4 -> 2, 7 -> 1)
```