How To Use Spark Transformations Efficiently For MapReduce-Like Jobs

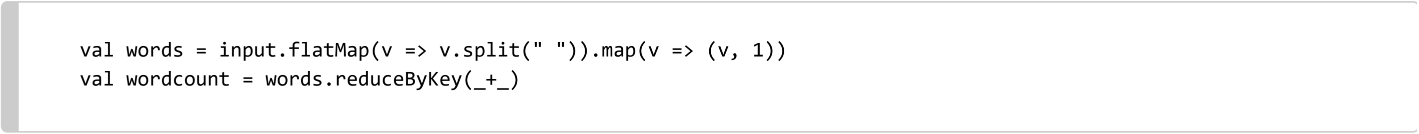
Author: Pengyu Wang

Apache Spark has become a popular platform for many types of big data workloads. The Spark API offers a well abstracted API that allows developers to easily solve big data problems. However, given the wide variety of transformations available with Spark, it's sometimes confusing which one delivers better performance for a particular use case. Like most organizations that used Spark for the first time, we enjoyed writing super simple code for our big data jobs. Yet, we were frustrated by job failures that were often difficult to diagnose.

When we developed MapReduce jobs, reduced phase bottleneck and potentially lower scalability were well understood. While MapReduce appears antiquated in comparison to Spark, MapReduce is surprisingly reliable and well behaved. With Spark, jobs can fail when transformations that require a data shuffle are used. This is often caused by large amounts of data piling up in internal data structures, resulting in out of memory exceptions or encountering YARN container constraints. Here we compare several commonly used transformations and provide tips that help can your job run reliably and efficiently with Spark.

reduceByKey

Most Spark documents recommend the **reduceByKey** transformation. The reason is that it implements a map side combiner which performs some aggregation in map side memory. This reduces the amount of shuffled data and avoids possible out of memory exceptions. Wordcount is a common example of **reduceByKey**:



```
val words = input.flatMap(v => v.split(" ")).map(v => (v, 1))
val wordcount = words.reduceByKey(_+_)
```

You might notice that in such use cases, each aggregation reduces two values into one by adding them up. The nature of reduceByKey places constraints on the aggregation operation. The aggregation operation must be additive, commutative, and associative, e.g. add, multiply, etc. For this reason, operations such as average and standard deviation cannot be directly implemented using **reduceByKey**.

groupByKey

groupByKey is an alternative for cases where the reduce operation is not additive, commutative, or associative. The **groupByKey** transformation aggregates all the values associated with each group and returns an **Iterable** for each collection. This is more flexible with the entire value set of the result resilient distributed dataset (RDD). So it is possible to perform any aggregation. Here is an example that calculates variance using **groupByKey**:

```
    val source = Array(("ABC", 10.5), ("ABC", 11.2), ("ABC", 10.7), ("ABC", 15.1), ("XYZ", 23.8), ("XYZ", 20.3), ("ABC", 12.5), ("XYZ", 25.9))
    val input2 = sc.parallelize(source)
    val variance = input2.groupByKey().map{case(k, v) => {
      var buffer = new ListBuffer[Double]

      // First pass: calculate mean
      var count = 0
      var sum = 0.0
      v.foreach(w => {
        buffer.+=(w)
        count += 1
        sum += w
      })
      var mean = sum / count

      // Second pass: calculate variance
      var error = 0.0
      v.foreach(w => {
        error += Math.pow(Math.abs(w - mean), 2)
      })
      (k, error)
}}
```

Using the **groupByKey** transformation, almost any aggregation can be achieved, especially if the output values are of a different type than the input. This expands the use case over **reduceByKey**. If**groupByKey** is followed by **flatMap**, it also supports multiple record output (no aggregation). However, the execution of **groupByKey** is memory intensive because all the values within a group are collected in a single in-memory partition after the shuffle. When key groups are large, out of memory exception are possible, so **groupByKey** should be avoided unless you're certain that key groups will fit in memory.

RepartitionAndSortWithinPartitions and MapPartitions

Spark also provides **mapPartitions** which performs a map operation on an entire partition. PairRDD's partitions are by default naturally based on physical HDFS blocks. But key grouping partitions can be created using **partitionBy** with a **HashPartitioner** class. This will push keys with same hashcode into the same partition, but without guaranteed ordering. Spark 1.2 introduced the **repartitionAndSortWithinPartitions** transformation which added sorting as part of the shuffle phase. This way, **mapPartitions** works on a key sorted partition, much like MapReduce. Unlike **groupByKey**, the records are not collected into memory all at once, but streamed from disk one record at a time using an iterator. The approach minimizes memory pressure.

repartitionAndSortWithinPartitions can also be used to perform "secondary sort". In many use cases, especially ETL processing, it is necessary to join multiple RDDs, or self-join a single RDD to attach group level attributes to a set of records. In the example below, the right-most fields of all the secondary records are modified to be identical to a primary record in the same group.

Data:

```
KEY_001,SECONDARY,V1,1
KEY_001,PRIMARY,V2,2
KEY_001, SECONDARY,V3,3
KEY_002, PRIMARY,V4,4
KEY_002,SECONDARY,V5,5
KEY_003, SECONDARY,V6,6
KEY_003, PRIMARY,V7,7
KEY_004, SECONDARY,V8,8
KEY_004, PRIMARY,V9,9
```

The partitioning group is identified, which is the key. In addition, the primary records are sorted with a priority. This requires composing a two part key by adding a sorting flag:

Key:

```
Primary: KEY_001|A
Secondary: KEY_001|B
```

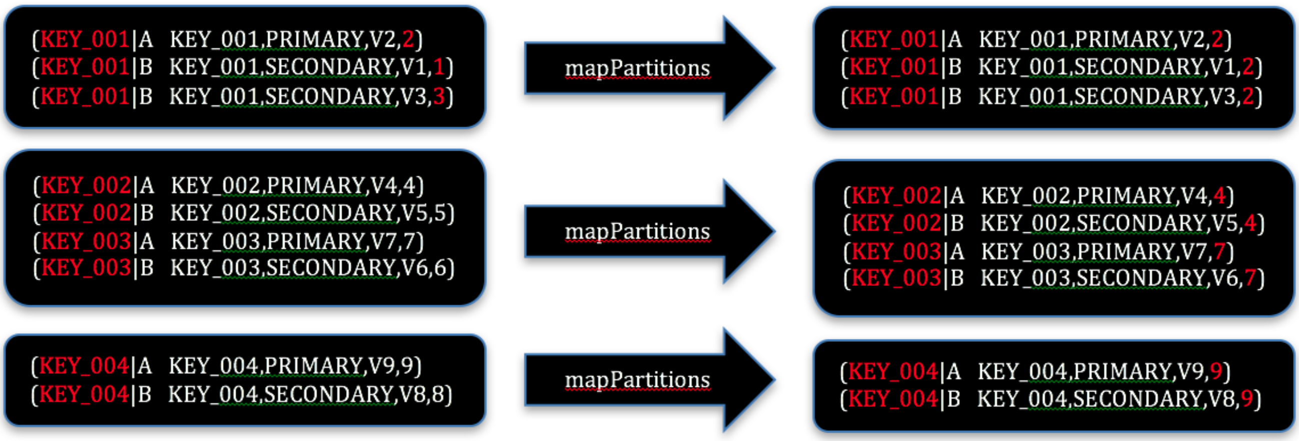
In the partitioner class, only the first part of the key is returned, which defines a group:

```
  class KeyBasePartitioner(partitions: Int) extends Partitioner {

    override def numPartitions: Int = partitions

    override def getPartition(key: Any): Int = {
      var k = key.asInstanceOf[String]
      var col = StringUtils.splitPreserveAllTokens(k, "|")
      Math.abs(col(0).hashCode() % numPartitions)
    }
  }
```

The RDD is then shuffled using **repartitionAndSortWithinPartitions** using the partitioner above. This produces a first part key based partitions where in each group primary records are always sorted on top of secondary. Due to the key hash collisions, a partition will have multiple keys in sorted order. So while iterating through resulting records, keep track of the key to detect when a group finishes and a new group starts.

```
 val partitionedRDD = inputRDD.repartitionAndSortWithinPartitions(new KeyBasePartitioner(100))

 val outputRDD = partitionedRDD.mapPartitions(v => {

   var outList = new ListBuffer[(String, String)]
   var lastKeyBase = null: String
   var attr = null: String
   while (v.hasNext) {
    var record = v.next
    // Parse key
    var key = record._1.split("\\|")
    var keyBase = key(0)
    var keyFlag = key(1)

    // Parse value
    var value = record._2.split(",")
    var inAttr = value(3)

    if (lastKeyBase != null && !lastKeyBase.equals(keyBase)) {
     // Reset group attribute
     attr = null
     lastKeyBase = keyBase
    }

    if (keyFlag.equals("A") && attr == null) {
     attr = inAttr
     counter += 1
    }

    var outRecord = (keyBase, value(0) + "," + value(1) + "," + value(2) + "," + (if (attr != null) attr else inAt
tr))
    outList.+=(outRecord)
   }

   outList.iterator
 })
```

At the output, the right most field in each record is modified to the value of the primary record. The **mapPartitions** transformation assumes the output can be more than one record. So the return value accepts an iterator of the output. This offers additional flexibility in reduce transformations.



```
(KEY_001|A  KEY_001,PRIMARY,V2,2)
(KEY_001|B  KEY_001,SECONDARY,V1,1)
(KEY_001|B  KEY_001,SECONDARY,V3,3)
```

```
(KEY_001|A  KEY_001,PRIMARY,V2,2)
(KEY_001|B  KEY_001,SECONDARY,V1,2)
(KEY_001|B  KEY_001,SECONDARY,V3,2)
```

```
(KEY_002|A  KEY_002,PRIMARY,V4,4)
(KEY_002|B  KEY_002,SECONDARY,V5,5)
(KEY_003|A  KEY_003,PRIMARY,V7,7)
(KEY_003|B  KEY_003,SECONDARY,V6,6)
```

```
(KEY_002|A  KEY_002,PRIMARY,V4,4)
(KEY_002|B  KEY_002,SECONDARY,V5,4)
(KEY_003|A  KEY_003,PRIMARY,V7,7)
(KEY_003|B  KEY_003,SECONDARY,V6,7)
```

```
(KEY_004|A  KEY_004,PRIMARY,V9,9)
(KEY_004|B  KEY_004,SECONDARY,V8,8)
```

```
(KEY_004|A  KEY_004,PRIMARY,V9,9)
(KEY_004|B  KEY_004,SECONDARY,V8,9)
```

Non-buffering Multiple Record Output with MapPartitions

Notice that during mapPartitions execution, an ArrayList of the entire record set is maintained until all the records are collected. With large datasets, again there may not be enough memory to hold the entire records, making the job fail. For cases like this, Spark RDDs do not directly support context write methods in the same way as MapReduce. But there is a workaround to output each record as they arrive without buffering, a custom iterator:

```
val partitionedRDD = inputRDD.repartitionAndSortWithinPartitions(new KeyBasePartitioner(100))

val outputRDD = partitionedRDD.mapPartitions(v => new CustomIterator(v))

class CustomIterator(iter: Iterator[(String, String)]) extends Iterator[(String, String)] {

  var lastKeyBase = null: String
  var attr = null: String

  def hasNext : Boolean = {
    iter.hasNext
  }

  def next : (String, String) = {
    var record = iter.next
    var key = record._1.split("\\|")
    var keyBase = key(0)
    var keyFlag = key(1)

    // Parse value
    var value = record._2.split(",")
    var inAttr = value(3)

    if (lastKeyBase != null && !lastKeyBase.equals(keyBase)) {
      // Reset group attribute
      attr = null
      lastKeyBase = keyBase
    }

    if (keyFlag.equals("A") && attr == null) {
      attr = inAttr
    }

    (keyBase, value(0) + "," + value(1) + "," + value(2) + "," + (if (attr != null) attr else inAttr))
  }
}
```

The CustomIterator class wraps an incoming iterator from **mapPartitions** and returned as the output of **mapPartitions**. The trick is to override the **next()** method to call the next() from the input iterator and handle any record manipulation logic. This way, records are streamed as they arrive and need be buffered in memory.

Conclusion

While Spark is a great step forward from MapReduce, it can fail if used improperly. Spark gets much of its speed through efficient use of memory. This is also its Achilles heel in cases where shuffled data does not fit in memory. Fortunately, there are techniques that can be used to avoid buffering large amounts of data in memory and avoid out of memory exceptions.
Using **repartitionAndSortWithinPartitions** and **mapPartitions** with custom iterators can solve these kinds of problems efficiently.