Return to book

Review this book

About the author

Knowledgebase

**1.** Best Practices

**1.1.** Avoid GroupByKey

**1.2.** Don't copy all elements of a la...

**1.3.** Gracefully Dealing with Bad In...

**2.** General Troubleshooting

**2.1.** Job aborted due to stage failu...

**2.2.** Missing Dependencies in Jar F...

**2.3.** Error running start-all.sh - Co...

**2.4.** Network connectivity issues b...

**3.** Performance & Optimization

**3.1.** How Many Partitions Does An...

**3.2.** Data Locality

**4.** Spark Streaming

**4.1.** ERROR OneForOneStrategy

# Avoid GroupByKey

Let's look at two different ways to compute word counts, one using `reduceByKey` and the other using `groupByKey`:

```
val words = Array("one", "two", "two", "three", "three", "three")
val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))

val wordCountsWithReduce = wordPairsRDD
  .reduceByKey(_ + _)
  .collect()

val wordCountsWithGroup = wordPairsRDD
  .groupByKey()
  .map(t => (t._1, t._2.sum))
  .collect()
```

While both of these functions will produce the correct answer, the `reduceByKey` example works much better on a large dataset. That's because Spark knows it can combine output with a common key on each partition before shuffling the data.

Look at the diagram below to understand what happens with `reduceByKey`. Notice how pairs on the same machine with the same key are combined (by using the lamdba function passed into `reduceByKey`) before the data is shuffled. Then the lamdba function is called again to reduce all the values from each partition to produce one final result.