Spark Solution for Rank Product

Mahmoud Parsian
Ph.D in Computer Science

Senior Architect @ illumina[1]

August 25, 2015

[1] www.illumina.com

Table of Contents

Outline

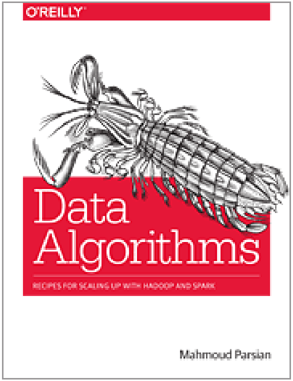
Who am I?

- Name: Mahmoud Parsian
- Education: Ph.D in Computer Science
- Work: Senior Architect @Illumina, Inc
  - Lead Big Data Team @Illumina
  - Develop scalable regression algorithms
  - Develop DNA-Seq and RNA-Seq workflows
  - Use Java/MapReduce/Hadoop/Spark/HBase
- Author: of 3 books
  - Data Algorithms (O'Reilly: http://shop.oreilly.com/product/0636920033950.do/)
  - JDBC Recipies (Apress: http://apress.com/)
  - JDBC MetaData Recipies (Apress: http://apress.com/))

Outline

Problem Statement: Data Algorithms Book

- Bonus Chapter: http://shop.oreilly.com/product/0636920033950.do
- Source code: https://github.com/mahmoudparsian/data-algorithms-book



Problem Statement

- Given a large set of studies, where each study is a set of (Key, Value) pairs
- The goal is to find the Rank Product of all given Keys for all studies.
- Magnitude of this data is challenging to store and analyze:
  - Several hundreds of studies
  - Each study has billions of (Key, Value) pairs
  - Find "rank product" for all given Keys

Problem Statement

- Given a large set of studies, where each study is a set of (Key, Value) pairs
- The goal is to find the Rank Product of all given Keys for all studies.
- Magnitude of this data is challenging to store and analyze:
  - Several hundreds of studies
  - Each study has billions of (Key, Value) pairs
  - Find "rank product" for all given Keys

Magnitude of Data per Analysis

- 100's of studies
- Each study may have billions of (K, V) pairs
- Example: 300 studies
- Each study: 2000,000,000 (K, V) pairs
- Analyze: 600,000,000,000 (K, V) pairs

Magnitude of Data per Analysis

- 100's of studies
- Each study may have billions of (K, V) pairs
- Example: 300 studies
- Each study: 2000,000,000 (K, V) pairs
- Analyze: 600,000,000,000 (K, V) pairs

Outline

Some Basic Definitions

- Study is a set of (K, V) pairs
- Example-1: K=geneID, V=geneValue
- Example-2: K=userID, V=user's followers in social networks
- Example-3: K=bookID, V=book rating by users

Some Basic Definitions

- Study is a set of (K, V) pairs
- Example-1: K=geneID, V=geneValue
- Example-2: K=userID, V=user's followers in social networks
- Example-3: K=bookID, V=book rating by users

What is a Ranking?

- Let S $= \{(K_1, 40), (K_2, 70), (K_3, 90), (K_4, 80)\}$
- Then Rank(S) $= \{(K_1, 4), (K_2, 3), (K_3, 1), (K_4, 2)\}$
- Since $90 > 80 > 70 > 40$
- Ranks are assigned as: 1, 2, 3, 4, ..., N

What is Rank Product?

- Let $\{A_1, ..., A_k\}$ be a set of (key-value) pairs where keys are unique per dataset.
- Example of (key-value) pairs:
  - (K,V) = (item, number of items sold)
  - (K,V) = (user, number of followers for the user)
  - (K,V) = (gene, test expression)
- Then the ranked product of $\{A_1, ..., A_k\}$ is computed based on the ranks r_i for key i across all k datasets. Typically ranks are assigned based on the sorted values of datasets.

What is Rank Product?

- Let $A_1 = \{(K_1, 30), (K_2, 60), (K_3, 10), (K_4, 80)\}$,
  then $Rank(A_1) = \{(K_1, 3), (K_2, 2), (K_3, 4), (K_4, 1)\}$
  since $80 > 60 > 30 > 10$
  Note that 1 is the highest rank (assigned to the largest value).
- Let $A_2 = \{(K_1, 90), (K_2, 70), (K_3, 40), (K_4, 50)\}$,
  $Rank(A_2) = \{(K_1, 1), (K_2, 2), (K_3, 4), (K_4, 3)\}$
  since $90 > 70 > 50 > 40$
- Let $A_3 = \{(K_1, 4), (K_2, 8)\}$
  $Rank(A_3) = \{(K_1, 2), (K_2, 1)\}$
  since $8 > 4$

The rank product of $\{A_1, A_2, A_3\}$ is expressed as:

$$\{(K_1, \sqrt[3]{3 \times 1 \times 2}), (K_2, \sqrt[3]{2 \times 2 \times 1}), (K_3, \sqrt[2]{4 \times 4}), (K_4, \sqrt[2]{1 \times 3})\}$$

Calculation of the Rank Product

- Given n genes and k replicates,
- Let $e_{g,i}$ be the fold change and $r_{g,i}$ the rank of gene g in the i'th replicate.
- Compute the rank product (RP) via the geometric mean:

$$RP(g) = \left(\prod_{i=1}^{k} r_{g,i} \right)^{1/k}$$

$$RP(g) = \sqrt[k]{\left(\prod_{i=1}^{k} r_{g,i} \right)}$$

Outline

Input Data Format

- Set of k studies $\{S_1, S_2, ..., S_k\}$
- Each study has billions of (Key, Value) pairs
- Sample Record:

  `<key-as-string><,><value-as-double-data-type>`

Input Data Persistence

- Data persists in HDFS
- Directory structure:

```
/input/study-001/file-001-1.txt
/input/study-001/file-001-2.txt
/input/study-001/file-001-3.txt
...
/input/study-235/file-235-1.txt
/input/study-235/file-235-2.txt
/input/study-235/file-235-3.txt
...
```

Formalizing Rank Product

- Let $S = \{S_1, S_2, ..., S_k\}$ be a set of k studies, where $k > 0$ and each study represent a micro-array experiment
- Let $S_i (i = 1, 2, ..., k)$ be a study, which has an arbitrary number of assays identified by $\{A_{i1}, A_{i2}, ...\}$
- Let each assay (can be represented as a text file) be a set of arbitrary number of records in the following format:

  `<gene_id><,><gene_value_as_double_data-type>`
- Let `gene_id` be in $\{g_1, g_2, ..., g_n\}$ (we have n genes).

Rank Product: in 2 Steps

Let $S = \{S_1, S_2, ..., S_k\}$ be a set of k studies:

- STEP-1: find the mean of values per study per gene
  - you may replace the "mean" function by your desired function
  - finding mean involves `groupByKey()` or `combineByKey()`

- STEP-2: perform the "Rank Product" per gene across all studies
  - finding "Rank Product" involves `groupByKey()` or `combineByKey()`

Formalizing Rank Product

The last step will be to find the rank product for each gene per study:

$$S_1 = \{(g_1, r_{11}), (g_2, r_{12}), \ldots\}$$

$$S_2 = \{(g_1, r_{21}), (g_2, r_{22}), \ldots\}$$

$$\ldots$$

$$S_k = \{(g_1, r_{k1}), (g_2, r_{k2}), \ldots\}$$

then Ranked Product of $g_j =$

$$RP(g_j) = \left(\prod_{i=1}^{k} r_{i,j} \right)^{1/k}$$

or

$$RP(g_j) = \sqrt[k]{\left(\prod_{i=1}^{k} r_{i,j} \right)}$$

Spark Solution for Rank Product

1. Read k input paths (each path represents a study, which may have any number of assay text files).
2. Find the mean per gene per study
3. Sort the genes by value per study and then assign rank values; To sort the dataset by value, we will swap the key with value and then perform the sort.
4. Assign ranks from 1, 2, ..., N (1 is assigned to the highest value) we use `JavaPairRDD.zipWithIndex()`, which zips the RDD with its element indices (these indices will be the ranks). Spark indices will start from 0, we will add 1 when computing the ranked product.
5. Finally compute the Rank Product per gene for all studies. This can be accomplished by grouping all ranks by the key (we may use `JavaPairRDD.groupByKey()` or `JavaPairRDD.combineByKey()` – note that, in general, `JavaPairRDD.combineByKey()` is more efficient than `JavaPairRDD.groupByKey()`).

Spark Solution for Rank Product

1. Read k input paths (each path represents a study, which may have any number of assay text files).
2. Find the mean per gene per study
3. Sort the genes by value per study and then assign rank values; To sort the dataset by value, we will swap the key with value and then perform the sort.
4. Assign ranks from 1, 2, ..., N (1 is assigned to the highest value) we use `JavaPairRDD.zipWithIndex()`, which zips the RDD with its element indices (these indices will be the ranks). Spark indices will start from 0, we will add 1 when computing the ranked product.
5. Finally compute the Rank Product per gene for all studies. This can be accomplished by grouping all ranks by the key (we may use `JavaPairRDD.groupByKey()` or `JavaPairRDD.combineByKey()` – note that, in general, `JavaPairRDD.combineByKey()` is more efficient than `JavaPairRDD.groupByKey()`).

Spark Solution for Rank Product

1. Read k input paths (each path represents a study, which may have any number of assay text files).
2. Find the mean per gene per study
3. Sort the genes by value per study and then assign rank values; To sort the dataset by value, we will swap the key with value and then perform the sort.
4. Assign ranks from 1, 2, ..., N (1 is assigned to the highest value) we use `JavaPairRDD.zipWithIndex()`, which zips the RDD with its element indices (these indices will be the ranks). Spark indices will start from 0, we will add 1 when computing the ranked product.
5. Finally compute the Rank Product per gene for all studies. This can be accomplished by grouping all ranks by the key (we may use `JavaPairRDD.groupByKey()` or `JavaPairRDD.combineByKey()` – note that, in general, `JavaPairRDD.combineByKey()` is more efficient than `JavaPairRDD.groupByKey()`).

Spark Solution for Rank Product

1. Read k input paths (each path represents a study, which may have any number of assay text files).
2. Find the mean per gene per study
3. Sort the genes by value per study and then assign rank values; To sort the dataset by value, we will swap the key with value and then perform the sort.
4. Assign ranks from 1, 2, ..., N (1 is assigned to the highest value) we use `JavaPairRDD.zipWithIndex()`, which zips the RDD with its element indices (these indices will be the ranks). Spark indices will start from 0, we will add 1 when computing the ranked product.
5. Finally compute the Rank Product per gene for all studies. This can be accomplished by grouping all ranks by the key (we may use `JavaPairRDD.groupByKey()` or `JavaPairRDD.combineByKey()` – note that, in general, `JavaPairRDD.combineByKey()` is more efficient than `JavaPairRDD.groupByKey()`).

Spark Solution for Rank Product

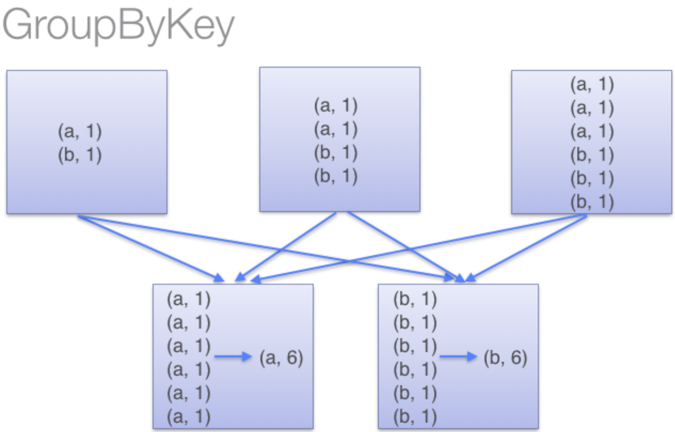
1. Read k input paths (each path represents a study, which may have any number of assay text files).
2. Find the mean per gene per study
3. Sort the genes by value per study and then assign rank values; To sort the dataset by value, we will swap the key with value and then perform the sort.
4. Assign ranks from 1, 2, ..., N (1 is assigned to the highest value) we use `JavaPairRDD.zipWithIndex()`, which zips the RDD with its element indices (these indices will be the ranks). Spark indices will start from 0, we will add 1 when computing the ranked product.
5. Finally compute the Rank Product per gene for all studies. This can be accomplished by grouping all ranks by the key (we may use `JavaPairRDD.groupByKey()` or `JavaPairRDD.combineByKey()` – note that, in general, `JavaPairRDD.combineByKey()` is more efficient than `JavaPairRDD.groupByKey()`).

Two Spark Solutions: groupByKey() and combineByKey()

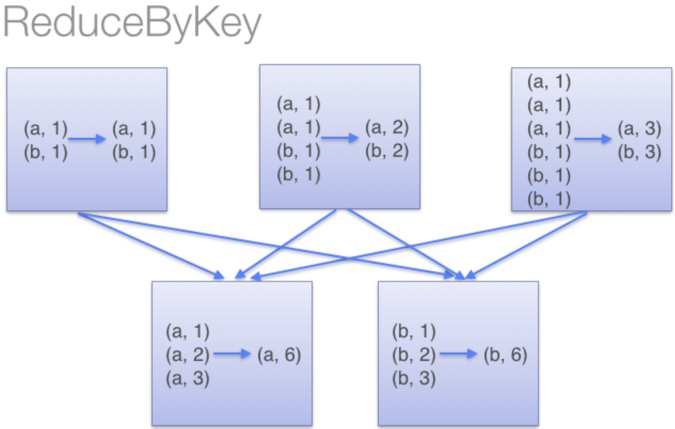
Two solutions are provided using Spark-1.4.0:

- `SparkRankProductUsingGroupByKey`
  - uses `groupByKey()`
- `SparkRankProductUsingCombineByKey`
  - uses `combineByKey()`

How does groupByKey() work



How does reduceByKey() work



Outline

Rank Product Algorithm in Spark

Algorithm: High-Level Steps	
Step	**Description**
STEP-1	import required interfaces and classes
STEP-2	handle input parameters
STEP-3	create a Spark context object
STEP-4	create list of studies (1, 2, ..., K)
STEP-5	compute mean per gene per study
STEP-6	sort by values
STEP-7	assign rank
STEP-8	compute rank product
STEP-9	save the result in HDFS

Main Driver

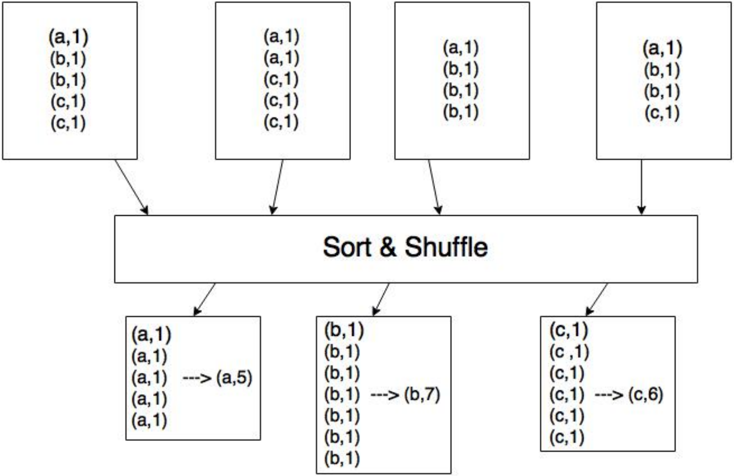
Listing 1: performRrankProduct()

```
public static void main(String[] args) throws Exception {
    // args[0] = output path
    // args[1] = number of studies (K)
    // args[2] =   input path for study 1
    // args[3] =   input path for study 2
    // ...
    // args[K+1] = input path for study K
    final String outputPath = args[0];
    final String numOfStudiesAsString = args[1];
    final int K = Integer.parseInt(numOfStudiesAsString);
    List<String> inputPathMultipleStudies = new ArrayList<String>();
    for (int i=1; i <= K; i++) {
        String singleStudyInputPath = args[1+i];
        inputPathMultipleStudies.add(singleStudyInputPath);
    }
    performRrankProduct(inputPathMultipleStudies, outputPath);
    System.exit(0);
}
```

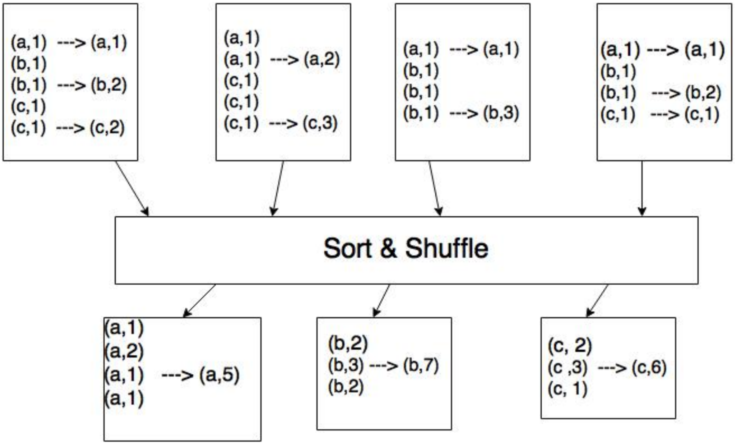
groupByKey() vs. combineByKey()

- Which one should we use? `combineByKey()` or `groupByKey()`? According to their semantics, both will give you the same answer. But `combineByKey()` is more efficient.
- In some situations, `groupByKey()` can even cause of out of disk problems. In general, `reduceByKey()`, and `combineByKey()` are preferred over `groupByKey()`.
- Spark shuffling is more efficient for `reduceByKey()` than `groupByKey()` and the reason is this: in the shuffle step for `reduceByKey()`, data is combined so each partition outputs at most one value for each key to send over the network, while in shuffle step for `groupByKey()`, all the data is wastefully sent over the network and collected on the reduce workers.
- To understand the difference, the following figures show how the shuffle is done for `reduceByKey()` and `groupByKey()`

Understanding groupByKey()



Understanding reduceByKey() or combineByKey()



Listing 2: performRrankProduct()

```java
public static void performRrankProduct(
                    final List<String> inputPathMultipleStudies,
                    final String outputPath)  throws Exception {
  // create a context object, which is used
  // as a factory for creating new RDDs
  JavaSparkContext context = Util.createJavaSparkContext(useYARN);

  // Spark 1.4.0 requires an array for creating union of many RDDs
  int index = 0;
  JavaPairRDD<String, Double>[] means =
        new JavaPairRDD[inputPathMultipleStudies.size()];
  for (String inputPathSingleStudy : inputPathMultipleStudies) {
     means[index] = computeMeanByGroupByKey(context, inputPathSingleStudy);
     index++;
  }

  // next compute rank
  ...
}
```

Listing 3: performRrankProduct()

```
public static void performRrankProduct(
  ...
  // next compute rank
  // 1. sort values based on absolute value of mean value
  // 2. assign rank from 1 to N
  // 3. calculate rank product for each gene
  JavaPairRDD<String,Long>[]  ranks = new JavaPairRDD[means.length];
  for (int i=0; i < means.length; i++) {
     ranks[i] = assignRank(means[i]);
  }
  // calculate ranked products
  // <gene, T2<rankedProduct, N>>
  JavaPairRDD<String, Tuple2<Double, Integer>> rankedProducts =
      computeRankedProducts(context, ranks);

  // save the result, shuffle=true
  rankedProducts.coalesce(1,true).saveAsTextFile(outputPath);

  // close the context and we are done
  context.close();
}
```

STEP-3: create a Spark context object

```java
public static JavaSparkContext createJavaSparkContext(boolean useYARN) {
  JavaSparkContext context;
  if (useYARN) {
      context = new JavaSparkContext("yarn-cluster", "MyAnalysis"); // YARN
  }
  else {
      context = new JavaSparkContext();  // Spark cluster
  }
  // inject efficiency
  SparkConf sparkConf = context.getConf();
  sparkConf.set("spark.kryoserializer.buffer.mb","32");
  sparkConf.set("spark.shuffle.file.buffer.kb","64");
  // set a fast serializer
  sparkConf.set("spark.serializer",
          "org.apache.spark.serializer.KryoSerializer");
  sparkConf.set("spark.kryo.registrator",
          "org.apache.spark.serializer.KryoRegistrator");
  return context;
}
```

Listing 4: STEP-4: compute mean

```
static JavaPairRDD<String, Double> computeMeanByGroupByKey(
          JavaSparkContext context,
          final String inputPath) throws Exception {
  JavaPairRDD<String, Double> genes =
     getGenesUsingTextFile(context, inputPath, 30);

  // group values by gene
  JavaPairRDD<String, Iterable<Double>> groupedByGene = genes.groupByKey();

  // calculate mean per gene
  JavaPairRDD<String, Double> meanRDD = groupedByGene.mapValues(
        new Function<
                     Iterable<Double>,      // input
                     Double                 // output: mean
                   >() {
        @Override
        public Double call(Iterable<Double> values) {
           double sum = 0.0;
           int count = 0;
           for (Double v : values) {
              sum += v;
              count++;
           }
           // calculate mean of samples
           double mean = sum / ((double) count);
           return mean;
        }
  });
  return meanRDD;
}
```

Listing 5: getGenesUsingTextFile()

```
static JavaPairRDD<String, Double> getGenesUsingTextFile(
        JavaSparkContext context,
        final String inputPath,
        final int numberOfPartitions) throws Exception {

  // read input and create the first RDD
  // JavaRDD<String>: where String = "gene,test_expression"
  JavaRDD<String> records = context.textFile(inputPath, numberOfPartitions);

  // for each record, we emit (K=gene, V=test_expression)
  JavaPairRDD<String, Double> genes
     = records.mapToPair(new PairFunction<String, String, Double>() {
        @Override
        public Tuple2<String, Double> call(String rec) {
            // rec = "gene,test_expression"
            String[] tokens = StringUtils.split(rec, ",");
            // tokens[0] = gene
            // tokens[1] = test_expression
            return new Tuple2<String, Double>(
               tokens[0], Double.parseDouble(tokens[1]));
        }
  });
  return genes;
}
```

Listing 6: Assign Rank

```
// result is JavaPairRDD<String, Long> = (gene, rank)
static JavaPairRDD<String,Long>  assignRank(JavaPairRDD<String,Double> rdd){
   // swap key and value (will be used for sorting by key); convert value to abs(value)
   JavaPairRDD<Double,String> swappedRDD = rdd.mapToPair(
            new PairFunction<Tuple2<String, Double>,       // T: input
                             Double,                       // K
                             String>>(){                   // V
            public Tuple2<Double,String> call(Tuple2<String, Double> s) {
                return new Tuple2<Double,String>(Math.abs(s._2), s._1);
            }
   });
   // we need 1 partition so that we can zip numbers into this RDD by zipWithIndex()
   JavaPairRDD<Double,String> sorted = swappedRDD.sortByKey(false, 1); // sort means descending
   // JavaPairRDD<T,Long> zipWithIndex()
   // Long values will be 0, 1, 2, ...; for ranking, we need 1, 2, 3, ..., therefore, we will add 1
   JavaPairRDD<Tuple2<Double,String>,Long> indexed = sorted.zipWithIndex();
   // next convert JavaPairRDD<Tuple2<Double,String>,Long> into JavaPairRDD<String,Long>
   //              JavaPairRDD<Tuple2<value,gene>,rank> into JavaPairRDD<gene,rank>
   JavaPairRDD<String, Long> ranked = indexed.mapToPair(
            new PairFunction<Tuple2<Tuple2<Double,String>,Long>,  // T: input
                             String,                              // K: mapped_id
                             Long>() {                            // V: rank
            public Tuple2<String, Long> call(Tuple2<Tuple2<Double,String>,Long> s) {
                return new Tuple2<String,Long>(s._1._2, s._2 + 1); // ranks are 1, 2, ..., n
            }
   });
   return ranked;
}
```

Listing 7: Compute Rank Product using groupByKey()

```java
static JavaPairRDD<String, Tuple2<Double, Integer>> computeRankedProducts(
        JavaSparkContext context,
        JavaPairRDD<String, Long>[] ranks) {
  JavaPairRDD<String, Long> unionRDD = context.union(ranks);

  // next find unique keys, with their associated values
  JavaPairRDD<String, Iterable<Long>> groupedByGeneRDD = unionRDD.groupByKey();

  // next calculate ranked products and the number of elements
  JavaPairRDD<String, Tuple2<Double, Integer>> rankedProducts = groupedByGeneRDD.mapValues(
       new Function<
                    Iterable<Long>,           // input: means for all studies
                    Tuple2<Double, Integer> // output: (rankedProduct, N)
                   >() {
       @Override
       public Tuple2<Double, Integer> call(Iterable<Long> values) {
          int N = 0;
          long products = 1;
          for (Long v : values) {
             products *= v;
             N++;
          }
          double rankedProduct = Math.pow( (double) products, 1.0/((double) N));
          return new Tuple2<Double, Integer>(rankedProduct, N);
       }
  });
  return rankedProducts;
}
```

Next FOCUS on combineByKey()

We do need to develop 2 functions:

- `computeMeanByCombineByKey()`
- `computeRankedProductsUsingCombineByKey()`

combineByKey(): how does it work?

- `combineByKey()` is the most general of the per-key aggregation functions. Most of the other per-key combiners are implemented using it.
- Like `aggregate()`, `combineByKey()` allows the user to return values that are not the same type as our input data.
- To understand `combineByKey()`, it is useful to think of how it handles each element it processes.
- As `combineByKey()` goes through the elements in a partition, each element either has a key it has not seen before or has the same key as a previous element.

combineByKey() definition

```
public <C> JavaPairRDD<K,C> combineByKey(
Function<V,C> createCombiner,     //   V -> C
Function2<C,V,C> mergeValue,      // C+V -> C
Function2<C,C,C> mergeCombiners   // C+C -> C
)
Description: Generic function to combine the elements for each key
using a custom set of aggregation functions. Turns a JavaPairRDD[(K, V)]
into a result of type JavaPairRDD[(K, C)], for a "combined type" C. Note
that V and C can be different -- for example, one might group an RDD of
type (Int, Int) into an RDD of type (Int, List[Int]).
Users provide three functions:
- createCombiner, which turns a V into a C (e.g., creates a one-element list)
- mergeValue, to merge a V into a C (e.g., adds it to the end of a list)
- mergeCombiners, to combine two Cs into a single one.
```

computeMeanByCombineByKey(): Define C data structure

```java
/**
 * AverageCount is used by combineByKey() to hold
 * the total values and their count.
 */
static class AverageCount implements Serializable {
    double total;
    int count;

    public AverageCount(double total, int count) {
        this.total = total;
        this.count = count;
    }

    public double average() {
        return total / (double) count;
    }
}
```

computeMeanByCombineByKey()

```java
static JavaPairRDD<String, Double> computeMeanByCombineByKey(
                        JavaSparkContext context,
                        final String inputPath) throws Exception {

   JavaPairRDD<String, Double> genes = getGenesUsingTextFile(context, inputPath, 30);

   // we need 3 function to be able to use combineByKey()
   Function<Double, AverageCount> createCombiner = ...
   Function2<AverageCount, Double, AverageCount> addAndCount = ...
   Function2<AverageCount, AverageCount, AverageCount> mergeCombiners = ...

   JavaPairRDD<String, AverageCount> averageCounts =
          genes.combineByKey(createCombiner, addAndCount, mergeCombiners);

   // now compute the mean/average per gene
   JavaPairRDD<String,Double> meanRDD = averageCounts.mapToPair(
            new PairFunction<
                             Tuple2<String, AverageCount>, // T: input
                             String,                       // K
                             Double                        // V
                            >() {
            @Override
            public Tuple2<String,Double> call(Tuple2<String, AverageCount> s) {
                return new Tuple2<String,Double>(s._1, s._2.average());
            }
   });
   return meanRDD;
}
```

3 functions for combineByKey()

```java
Function<Double, AverageCount> createCombiner = new Function<Double, AverageCount>() {
  @Override
  public AverageCount call(Double x) {
     return new AverageCount(x, 1);
  }
};

Function2<AverageCount, Double, AverageCount> addAndCount =
   new Function2<AverageCount, Double, AverageCount>() {
   @Override
   public AverageCount call(AverageCount a, Double x) {
      a.total += x;
      a.count += 1;
      return a;
   }
};

Function2<AverageCount, AverageCount, AverageCount> mergeCombiners =
   new Function2<AverageCount, AverageCount, AverageCount>() {
   @Override
   public AverageCount call(AverageCount a, AverageCount b) {
     a.total += b.total;
     a.count += b.count;
     return a;
   }
};
```

Per-key average using combineByKey() in Python

```
sumCount = nums.combineByKey(
                (lambda x: (x,1)),
                (lambda x, y: (x[0] + y, x[1] + 1)),
                (lambda x, y: (x[0] + y[0], x[1] + y[1]))
            )

mean = sumCount.map(lambda key, xy: (key, xy[0]/xy[1])).collectAsMap()
print(*mean)
```

Sample Run using groupByKey(): script

```
# define input/output for Hadoop/HDFS
#// args[0] = output path
#// args[1] = number of studies (K)
#// args[2] =   input path for study 1
#// args[3] =   input path for study 2
#// ...
#// args[K+1] = input path for study K
OUTPUT=/rankproduct/output
NUM_OF_INPUT=3
INPUT1=/rankproduct/input1
INPUT2=/rankproduct/input2
INPUT3=/rankproduct/input3

# remove all files under input
$HADOOP_HOME/bin/hadoop fs -rmr $OUTPUT
#
# remove all files under output
driver=org.dataalgorithms.bonus.rankproduct.spark.SparkRankProductUsingGroupByKey
$SPARK_HOME/bin/spark-submit --class $driver \
    --master yarn-cluster \
    --jars $OTHER_JARS \
    --conf "spark.yarn.jar=$SPARK_JAR" \
    $APP_JAR $OUTPUT $NUM_OF_INPUT $INPUT1  $INPUT2  $INPUT3
```

Sample Run using groupByKey(): input

```
# hadoop fs -cat /rankproduct/input1/rp1.txt
K_1,30.0
K_2,60.0
K_3,10.0
K_4,80.0

# hadoop fs -cat /rankproduct/input2/rp2.txt
K_1,90.0
K_2,70.0
K_3,40.0
K_4,50.0

# hadoop fs -cat /rankproduct/input3/rp3.txt
K_1,4.0
K_2,8.0
```

Sample Run using groupByKey(): output

```
# hadoop fs -cat /rankproduct/output/part*
(K_2,(1.5874010519681994,3))
(K_3,(4.0,2))
(K_1,(1.8171205928321397,3))
(K_4,(1.7320508075688772,2))
```

computeRankedProductsUsingCombineByKey(): Define C data structure

```
/**
 * RankProduct is used by combineByKey() to hold
 * the total product values and their count.
 */
static class RankProduct implements Serializable {
    long product;
    int count;

    public RankProduct(long product, int count) {
        this.product = product;
        this.count = count;
    }

    public double rank() {
        return Math.pow((double) product, 1.0/ (double) count);
    }
}
```

computeRankedProductsUsingCombineByKey()

```java
// JavaPairRDD<String, Tuple2<Double, Integer>> = <gene, T2(rankedProduct, N>>
// where N is the number of elements for computing the rankedProduct
static JavaPairRDD<String, Tuple2<Double, Integer>> computeRankedProductsUsingCombineByKey(
      JavaSparkContext context,
      JavaPairRDD<String, Long>[] ranks) {
  JavaPairRDD<String, Long> unionRDD = context.union(ranks);

  // we need 3 function to be able to use combinebyKey()
  Function<Long, RankProduct> createCombiner = ...
  Function2<RankProduct, Long, RankProduct> addAndCount = ...
  Function2<RankProduct, RankProduct, RankProduct> mergeCombiners = ...

  // next find unique keys, with their associated copa scores
  JavaPairRDD<String, RankProduct> combinedByGeneRDD =
     unionRDD.combineByKey(createCombiner, addAndCount, mergeCombiners);

  // next calculate ranked products and the number of elements
  JavaPairRDD<String, Tuple2<Double, Integer>> rankedProducts = combinedByGeneRDD.mapValues(
          new Function<
                        RankProduct,               // input: RankProduct
                        Tuple2<Double, Integer>    // output: (RankedProduct.count)
                       >() {
          @Override
          public Tuple2<Double, Integer> call(RankProduct value) {
              double theRankedProduct = value.rank();
              return new Tuple2<Double, Integer>(theRankedProduct, value.count);
          }
  });
  return rankedProducts;
}
```

3 functions for computeRankedProductsUsingCombineByKey()

```
Function<Long, RankProduct> createCombiner = new Function<Long, RankProduct>() {
     @Override
     public RankProduct call(Long x) {
       return new RankProduct(x, 1);
     }
};

Function2<RankProduct, Long, RankProduct> addAndCount =
     new Function2<RankProduct, Long, RankProduct>() {
     @Override
     public RankProduct call(RankProduct a, Long x) {
       a.product *= x;
       a.count += 1;
       return a;
     }
};

Function2<RankProduct, RankProduct, RankProduct> mergeCombiners =
     new Function2<RankProduct, RankProduct, RankProduct>() {
     @Override
     public RankProduct call(RankProduct a, RankProduct b) {
       a.product *= b.product;
       a.count += b.count;
       return a;
     }
};
```

REVISED computeRankedProductsUsingCombineByKey(): Define C data structure

```
static class RankProduct implements Serializable {
    long product;
    int count;

    public RankProduct(long product, int count) {
        this.product = product;
        this.count = count;
    }

    public product(long value) {
       this.product *= value;
       this.count++;
    }

    public product(RankProduct pr) {
       this.product *= pr.value;
       this.count += pr.count;
    }

    public double rank() {
        return Math.pow((double) product, 1.0/ (double) count);
    }
}
```

REVISED 3 functions for computeRankedProductsUsingCombineByKey()

```
Function<Long, RankProduct> createCombiner = new Function<Long, RankProduct>() {
     public RankProduct call(Long x) {
       return new RankProduct(x, 1);
     }
};

Function2<RankProduct, Long, RankProduct> addAndCount =
     new Function2<RankProduct, Long, RankProduct>() {
     public RankProduct call(RankProduct a, Long x) {
       a.product(x);
       return a;
     }
};

Function2<RankProduct, RankProduct, RankProduct> mergeCombiners =
     new Function2<RankProduct, RankProduct, RankProduct>() {
     public RankProduct call(RankProduct a, RankProduct b) {
       a.product(b);
       return a;
     }
};
```

Outline

Moral of Story...

- Understand data requirements
- Select proper platform for implementation: Spark
- Partition your RDDs properly
  - number of cluster nodes
  - number of cores per node
  - amount of RAM
- Avoid unnecessary computations (g1, g2), (g2, g1)
- Use filter() often to remove non-needed RDD elements
- Avoid unnecessary RDD.saveAsTextFile(path)
- Run both on YARN and Spark cluster and compare performance
- use `groupByKey()` cautiously
- use `combineByKey()` over `groupByKey()`
- Verify your test results against R language

Thank you!

Questions?