Cancer Outlier Profile Analysis using Apache Spark

Mahmoud Parsian
Ph.D in Computer Science
Senior Architect @ illumina

Apache Big Data
Vancouver, BC

May 10, 2016
Outline

1 Biography

2 What is COPA using Spark?

3 COPA Algorithm

4 Input Data

5 Rank Product Algorithm

6 Moral of Story

7 References
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 Recipes (Apress: http://apress.com/)
  - JDBC MetaData Recipes (Apress: http://apress.com/)
My book: Data Algorithms

Outline

1 Biography

2 What is COPA using Spark?

3 COPA Algorithm

4 Input Data

5 Rank Product Algorithm

6 Moral of Story

7 References
What is COPA using Spark?

Background: COPA, Spark

- What is COPA? Cancer Outlier Profile Analysis
- Is COPA important?
- What is an outlier?
- Common methods to detect outliers
- Outliers in cancer gene study
- T-statistic in outlier study
- What are COPA algorithms?
What is COPA?

- **COPA algorithm**: analyze DNA microarray data for genes markedly over-expressed (outliers) in a subset of cases.
- COPA: first algorithm that lead to the discovery of ERG rearrangement in prostate cancer.
- ERG (ETS-related gene) is an oncogene meaning that it encodes a protein that typically is mutated in cancer.
- COPA identified the ETS family members ERG and ETV1 as high-ranking outliers in multiple prostate cancer profiling studies.
What is COPA using Spark?

Informal COPA Algorithm

**Input** = A cancer gene expression dataset, consisting of thousands of genes measured across hundreds of samples

**STEPS:** 

(A) → (B) → (C) → (D) → (E)

(A) genes are normalized and sorted via the COPA method.

(B)  
- COPA normalizes the **median expression** per gene to zero
- The **MAD (median absolute deviation)** to one
- A **percentile cutoff** is selected (e.g., 75th%, 90th% or 95th%) and genes are sorted by their COPA value at the selected percentile

(C) **The top 1%** of genes is deemed to have outlier expression profiles

(D) ...

(E) ...
What is COPA using Spark?

Informal COPA Algorithm

**STEPS:**  
(A) → (B) → (C) → (D) → (E)

- A, B, C

**D**  
- A **collection of outliers** from independent datasets are compiled and submitted for meta-analysis

**E**  
- Multiple datasets of a given cancer type are meta-analyzed
- Identify genes that are **consistently called outliers** across independent datasets
- Significance is assessed via the binomial distribution
What is an Outlier?

- **Definition:**
  - **outlier** is an *unusual value* in a dataset;
  - one that does not fit the typical pattern of data

- **Sources of outliers**
  - Recording of measurement errors
  - Natural variation of the data (valid data)
What is an Outlier?

[Graph showing a scatter plot with a point significantly isolated from the main cluster of data points.]
What is COPA Statistics?

- COPA = Cancer Outlier Profile Analysis
- Statistics designed to identify outliers in cancer gene expression profile

ACTL8 (FC=1.27 FDR=0.031)
Outlier Detection, How?

- Visual inspection of data
  - not applicable for large complex datasets

- Automated methods
  - Normal distribution-based method (NORMAL)
  - Median Absolute Deviation (MAD)
  - Distance Based Method
  - COPA
  - ...
Normal distribution-based method

\[ f(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-(x-\mu)^2 / 2\sigma^2} \]

\[ \int_{-\infty}^{\infty} f(x) \, dx = 1 \]

where

- \( x \): sample data
- \( \mu \): mean of distribution
- \( \sigma \): standard deviation
- \( \sigma^2 \): variance
What is COPA using Spark?

Normal distribution in picture...

99.7% of the data are within 3 standard deviations of the mean

95% within 2 standard deviations

68% within 1 standard deviation
Normal distribution -based method

- Assume normal distribution for each variable
- Works on one variable at a time: $X_k, k = 1, ..., p$
- Algorithm
  - Let $i$’th sample data for variable $X_k, (i = 1, ..., n)$: $x_{ik}$
  - Sample mean for $X_k$: $\mu = \frac{1}{n} \sum_{i=1}^{n} (x_{ik})$
  - Sample standard deviation for $X_k$: $S_k = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_{ik} - \mu)^2}$
  - Calculate spatial statistics $Z_{ik}$ for $i = 1, ..., n$ as $Z_{ik} = \frac{x_{ik} - \mu}{S_k}$
  - Label $x_{ik}$ an outlier if $|Z_{ik}| > 3$
Problems with Normal distribution-based method

- Very dependent on assumption of normality
- Does not work well on big data
- Works only for numeric-based variables
- $\mu$ (mean) and $S_k$ (standard deviation) are not robust to outliers
Median Absolute Deviation (MAD)

- Another method for dealing with robustness problem
- Use median as robust estimate of mean
- Use MAD as robust estimate of standard deviation

Algorithm

- Calculate $D_{ik} = |X_{ik} - \text{median}(X_k)|$ for $i=1,...,n$
- Calculate $\text{MAD} = \text{median}(D_{1k},...,D_{nk})$
- Calculate spatial statistics $Z_{ik} = \frac{x_{ik} - \text{median}(X_k)}{1.4826 \times \text{MAD}}$
- Label $x_{ik}$ an outlier if $|Z_{ik}| > 3.5$
- Note: standard deviation $\sigma \approx 1.4826 \times \text{MAD}$
Outliers in Cancer Studies

- In cancer studies:
  - mutations can often amplify or turn off gene expression in only a minority of samples, namely produce outliers.
  - t-statistic may yield high false discovery rate (FDR) when trying to detect changes occurred in a small number of samples.

- COPA

- Is there a better method than COPA? may be?
Outline

1. Biography
2. What is COPA using Spark?
3. COPA Algorithm
4. Input Data
5. Rank Product Algorithm
6. Moral of Story
7. References
COPA Algorithm

- Gene expression values are median centered.
- Setting each gene’s median expression value to zero.
- The median absolute deviation (MAD) is calculated.
- Scaled to 1 by dividing each gene expression value by its MAD.
- The 75th, 90th, and 95th percentiles of the transformed expression values are tabulated for each gene.
- Genes are rank-ordered by their percentile scores, providing a prioritized list of outlier profiles.
Median and MAD were used for transformation as opposed to mean and standard deviation so that outlier expression values do not unduly influence the distribution estimates, and are thus preserved post-normalization.
COPA: ONE-Sided and TWO-Sided

- **ONE-Sided**
  - Put all Studies together
  - Calculate COPA-Score per GENE-ID

- **TWO-Sided**
  - Create K Study groups
  - **Rank All Groups**
  - Calculate COPA-Score per GENE-ID (per Study Group)
COPA: TWO-Sided

- Handles K studies, where each study is a set of samples.
- For each study, find the COPA scores and then the result will be the median of the COPA scores for all studies.
- For K studies, report
  - GENE-ID\(_i\)
  - mean(COPA scores)
  - geometric mean of product rank \((j=1, \ldots, K)\)
  - Rank-Converted-Scores\(_{ij}\) for \((j=1, \ldots, K)\)
Is COPA Big Data?

- **General case:**
  - 1000’s of Studies
  - Each Study = 100’s of Samples
  - Each Sample = 100,000+ (K, V) pairs

- **Example:**
  - 4000 studies, assume each study has 300 samples
  - Total Samples: $4000 \times 300 = 1,200,000$
  - Analyze: $1,200,000 \times 100,000 = 120,000,000,000$ (K, V) pairs
Input Format

- Each sample is a set of (K, V) pairs
- General Format: K=GENE-ID, V=GENE-VALUE
- Example-1: g127, 1.20
- Example-2: g4567, 0.09
Input Format

- Each sample is a set of (K, V) pairs
- General Format: \( K=\text{GENE-ID}, \ V=\text{GENE-VALUE} \)
- Example-1: g127,1.20
- Example-2: g4567,0.09
Outline

1 Biography
2 What is COPA using Spark?
3 COPA Algorithm
4 Input Data
5 Rank Product Algorithm
6 Moral of Story
7 References
Input Data Format

- Set of k studies \( \{S_1, S_2, \ldots, S_k\} \)
- Each study has billions of (Key, Value) pairs
- Sample Record:
  
  \(<\text{key-as-string}>,<\text{value-as-double-data-type}>\)
Input Data Persistence

- Data persists in HDFS
- Directory structure:
  - /input/study-0001/file-0001-1.txt
  - /input/study-0001/file-0001-2.txt
  - /input/study-0001/file-0001-3.txt
  ...
  - /input/study-3265/file-3265-1.txt
  - /input/study-3265/file-3265-2.txt
  - /input/study-3265/file-3265-3.txt
  ...

Mahmoud Parsian Ph.D in Computer Science Senior Architect @ Illumina

Cancer Outlier Profile Analysis using Apache Spark
What is a Ranking?

- Let $S = \{(K_1, 40), (K_2, 70), (K_3, 90), (K_4, 80)\}$
- Then $\text{Rank}(S) = \{(K_1, 4), (K_2, 3), (K_3, 1), (K_4, 2)\}$
- Since $90 > 80 > 70 > 40$
- Ranks are assigned as: $1, 2, 3, 4, \ldots, N$
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 $\text{Rank}(A_1) = \{(K_1, 3), (K_2, 2), (K_3, 4), (K_4, 1)\}$ since $80 > 60 > 30 > 10$
  
  Note that 1 is the highest rank (assigned to the largest value).

- Let $A_2 = \{(K_1, 90), (K_2, 70), (K_3, 40), (K_4, 50)\}$,
  
  $\text{Rank}(A_2) = \{(K_1, 1), (K_2, 2), (K_3, 4), (K_4, 3)\}$ since $90 > 70 > 50 > 40$

- Let $A_3 = \{(K_1, 4), (K_2, 8)\}$
  
  $\text{Rank}(A_3) = \{(K_1, 2), (K_2, 1)\}$ 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:

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

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

- Let $S = \{S_1, S_2, \ldots, S_k\}$ be a set of $k$ studies, where $k > 0$ and each study represent a micro-array experiment.
- Let $S_i (i = 1, 2, \ldots, k)$ be a study, which has an arbitrary number of assays identified by $\{A_i_1, A_i_2, \ldots\}$.
- 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 $\{g_1, g_2, \ldots, g_n\}$ (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:

\[ S_1 = \{(g_1, r_{11}), (g_2, r_{12}), \ldots\} \]
\[ S_2 = \{(g_1, r_{21}), (g_2, r_{22}), \ldots\} \]

...  
\[ S_k = \{(g_1, r_{k1}), (g_2, r_{k2}), \ldots\} \]

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 (path = study)
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)
   - 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
5. Finally compute the Rank Product per gene for all studies:
   - JavaPairRDD.groupByKey() (not efficient)
   - JavaPairRDD.combineByKey() (efficient)
Spark Solution for Rank Product

1. Read k input paths (path = study)
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)
   - 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
5. Finally compute the Rank Product per gene for all studies:
   - JavaPairRDD.groupByKey() (not efficient)
   - JavaPairRDD.combineByKey() (efficient)
Spark Solution for Rank Product

1. Read k input paths (path = study)
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)
   - 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
5. Finally compute the Rank Product per gene for all studies:
   - JavaPairRDD.groupByKey() (not efficient)
   - JavaPairRDD.combineByKey() (efficient)
Spark Solution for Rank Product

1. Read $k$ input paths ($\text{path} = \text{study}$)
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)
   - use $\text{JavaPairRDD.zipWithIndex}()$, which zips the RDD with its element indices (these indices will be the ranks).
   - Spark indices will start from 0, we will add 1
5. Finally compute the Rank Product per gene for all studies:
   - $\text{JavaPairRDD.groupByKey}()$ (not efficient)
   - $\text{JavaPairRDD.combineByKey}()$ (efficient)
Spark Solution for Rank Product

1. Read k input paths \( \text{path} = \text{study} \)
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)
   - use \text{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
5. Finally compute the Rank Product per gene for all studies:
   - \text{JavaPairRDD.groupByKey()} (not efficient)
   - \text{JavaPairRDD.combineByKey()} (efficient)
Two Spark Solutions: `groupByKey()` and `combineByKey()`

Two solutions are provided using Spark-1.6.0:

- **SparkRankProductUsingGroupByKey**
  - uses `groupByKey()`

- **SparkRankProductUsingCombineByKey**
  - uses `combineByKey()`
How does `groupByKey()` work

**GroupByKey**

- Input Data
  - (a, 1)
  - (b, 1)
- Grouped Data
  - (a, 1)
  - (a, 1)
  - (a, 1)
  - (a, 1)
  - (b, 1)
  - (b, 1)
- Output Data
  - (a, 6)
  - (b, 6)
How does `reduceByKey()` work

**ReduceByKey**

Input Data

Mahmoud Parsian  Ph.D in Computer Science  Senior Architect @ Illumina

Cancer Outlier Profile Analysis using Apache Spark
Outline

1. Biography
2. What is COPA using Spark?
3. COPA Algorithm
4. Input Data
5. Rank Product Algorithm
6. Moral of Story
7. References
## Rank Product Algorithm in Spark

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEP-1</td>
<td>import required interfaces and classes</td>
</tr>
<tr>
<td>STEP-2</td>
<td>handle input parameters</td>
</tr>
<tr>
<td>STEP-3</td>
<td>create a Spark context object</td>
</tr>
<tr>
<td>STEP-4</td>
<td>create list of studies (1, 2, ..., K)</td>
</tr>
<tr>
<td>STEP-5</td>
<td>compute mean per gene per study</td>
</tr>
<tr>
<td>STEP-6</td>
<td>sort by values</td>
</tr>
<tr>
<td>STEP-7</td>
<td>assign rank</td>
</tr>
<tr>
<td>STEP-8</td>
<td>compute rank product</td>
</tr>
<tr>
<td>STEP-9</td>
<td>save the result in HDFS</td>
</tr>
</tbody>
</table>
Main Driver

Listing 1: `performRrankProduct()`

```java
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 1.6.0 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()

```java
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

```java
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>, Double
            // input
            // 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;
    }
```

Mahmoud Parsian Ph.D in Computer Science Senior Cancer Outlier Profile Analysis using Apache Spark
```java
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

```java
// 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>, Double, String>() {
            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<Double, String>, Long> indexed = sorted.zipWithIndex();
    // JavaPairRDD<Tuple2<Double, String>,Long> into JavaPairRDD<String,Long>
    JavaPairRDD<Tuple2<Double, String>, Long> ranked = indexed.mapToPair(
        new PairFunction<Tuple2<Tuple2<Double, String>, Long>, String, Long>() {
            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()

```
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>,
            Tuple2<Double, Integer>>()
            { @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() by Combiner (C)

Provide 3 functions:

1. createCombiner, which turns a V into a C
   // creates a one-element list

2. mergeValue, to merge a V into a C
   // adds it to the end of a list

3. 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;
    }
}
```
```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>, String, Double>() {

            @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

```python
# C = Tuple2(sum, count)

numscombineByKey =
(lambda v: (v,1)),
(lambda C, v: (C[0] + v, C[1] + 1)),
(lambda C1, C2: (C1[0] + C2[0], C1[1] + C2[1]))

mean = sumCount.map(lambda key, C: (key, C[0]/C[1])).collectAsMap()

print(*mean)
```
Sample Run using groupByKey(): script

```java
# 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

```bash
1  # hadoop fs -cat /rankproduct/input1/rp1.txt
2  K_1,30.0
3  K_2,60.0
4  K_3,10.0
5  K_4,80.0
6  
7  # hadoop fs -cat /rankproduct/input2/rp2.txt
8  K_1,90.0
9  K_2,70.0
10 K_3,40.0
11 K_4,50.0
12
13 # hadoop fs -cat /rankproduct/input3/rp3.txt
14 K_1,4.0
15 K_2,8.0
```
Sample Run using `groupByKey()`: output

```bash
# 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

```java
/**
 * 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,
            Tuple2<Double, Integer>
        >() {
            @Override
            public Tuple2<Double, Integer> call(RankProduct value) {
                double theRankedProduct = value.rank();
                return new Tuple2<Double, Integer>(theRankedProduct, value.count);
            }
        });
    return rankedProducts;
}
```

Mahmoud Parsian Ph.D in Computer Science Senior Architect @ Illumina
Cancer Outlier Profile Analysis using Apache Spark 66 / 74
3 functions for computeRankedProductsUsingCombineByKey()

```java
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;
        }
    };
```
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()

```java
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

1. Biography
2. What is COPA using Spark?
3. COPA Algorithm
4. Input Data
5. Rank Product Algorithm
6. Moral of Story
7. References
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 \((g_1, g_2), (g_2, g_1)\)
- Use \(\text{filter()}\) often to remove non-needed RDD elements
- Avoid unnecessary RDD.saveAsTextFile(path)
- Run both on YARN and Spark cluster and compare performance
- use \(\text{groupByKey()}\) cautiously
- use \(\text{combineByKey()}\) over \(\text{groupByKey()}\)
- Verify your test results against R language
Outline

1. Biography
2. What is COPA using Spark?
3. COPA Algorithm
4. Input Data
5. Rank Product Algorithm
6. Moral of Story
7. References
References

Thank you!

Questions?