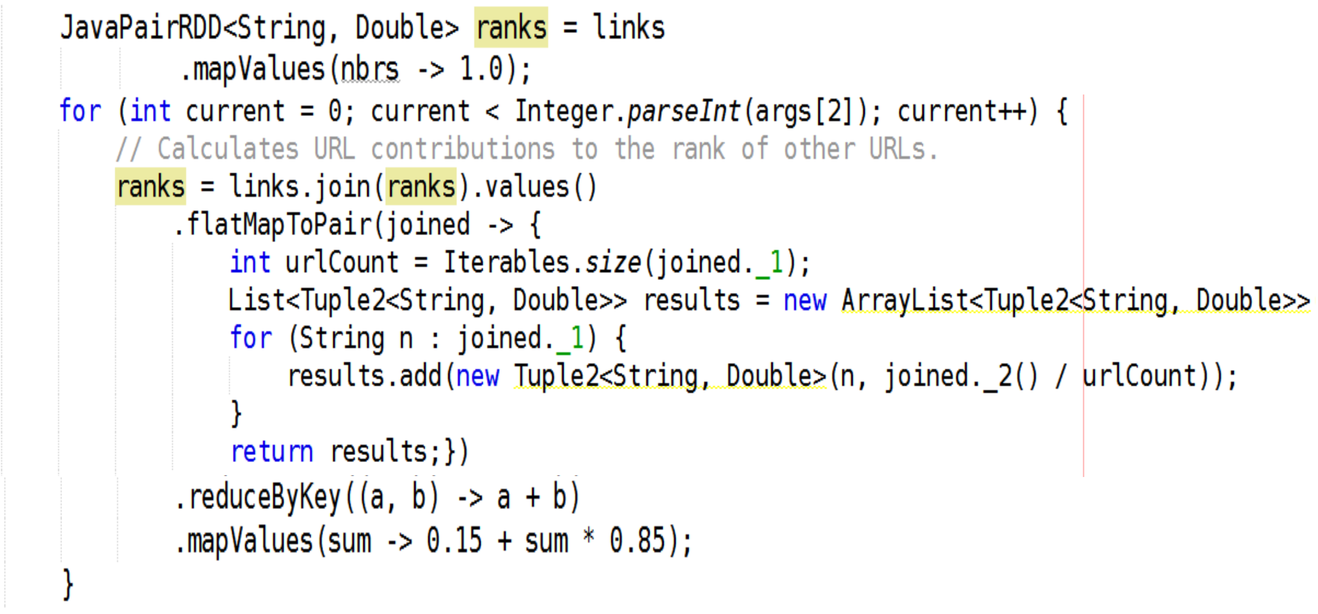
Optimized Reduce Side Join
By Mohit Kumar

Spark Internals:Pagerank

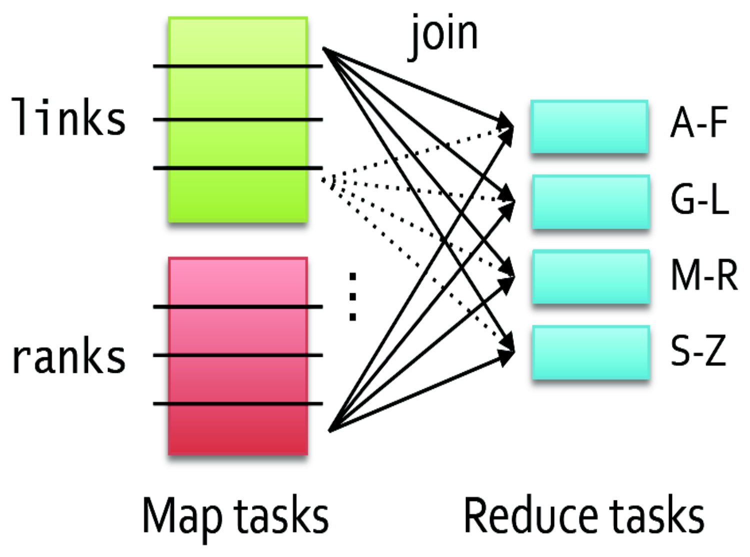


```
JavaPairRDD<String, Double> ranks = links
        .mapValues(nbrs -> 1.0);
for (int current = 0; current < Integer.parseInt(args[2]); current++) {
    // Calculates URL contributions to the rank of other URLs.
    ranks = links.join(ranks).values()
        .flatMapToPair(joined -> {
            int urlCount = Iterables.size(joined._1);
            List<Tuple2<String, Double>> results = new ArrayList<Tuple2<String, Double>>
            for (String n : joined._1) {
                results.add(new Tuple2<String, Double>(n, joined._2() / urlCount));
            }
            return results;})
        .reduceByKey((a, b) -> a + b)
        .mapValues(sum -> 0.15 + sum * 0.85);
}
```

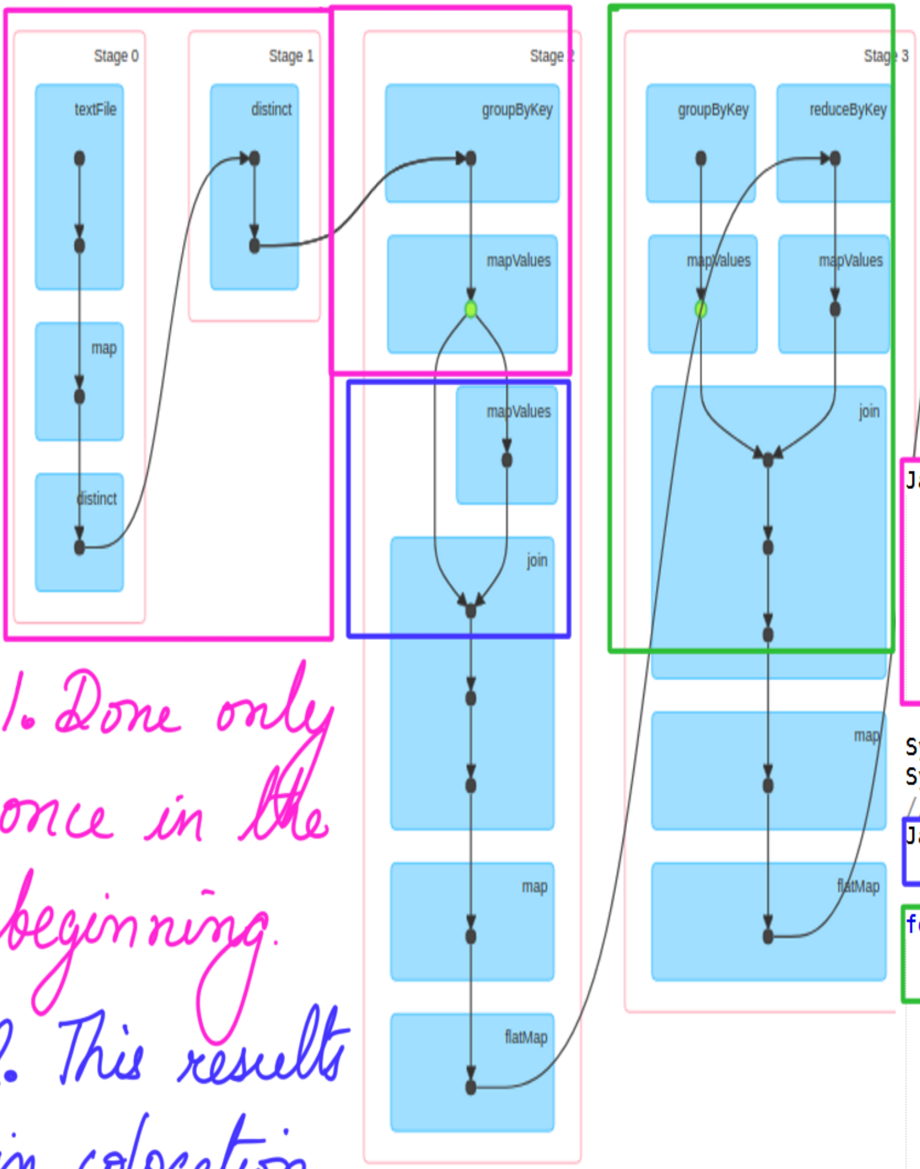
`links` and `ranks` are repeatedly joined

Each join requires a full shuffle over the network

» Hash both onto same nodes



Spark Internals:Pagerank

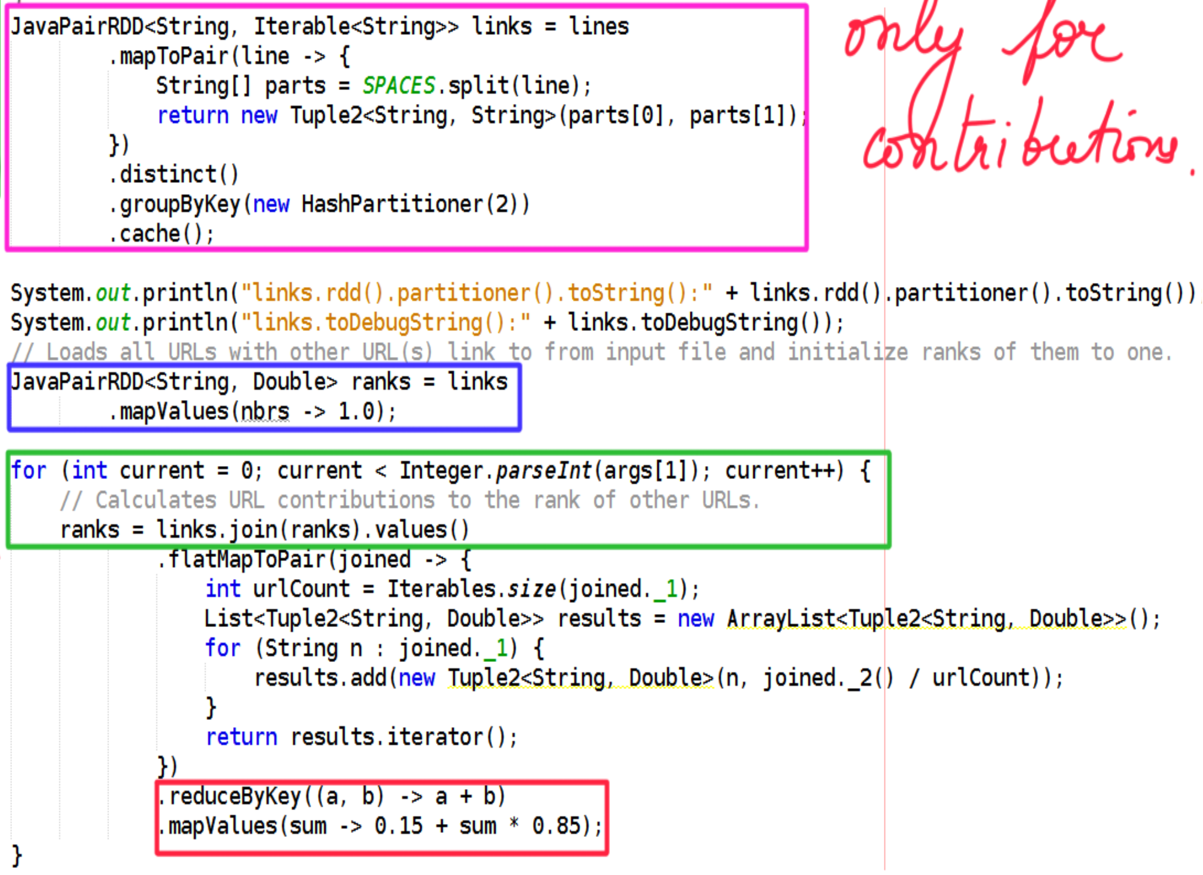


1. Done only once in the beginning.

2. This results in colocation and copartitioning

3. Repeated join between 2 huge datasets but without shuffle and sort.

4. Price of shuffle sort paid but only for contributions.



```
JavaPairRDD<String, Iterable<String>> links = lines
        .mapToPair(line -> {
            String[] parts = SPACES.split(line);
            return new Tuple2<String, String>(parts[0], parts[1]);
        })
        .distinct()
        .groupByKey(new HashPartitioner(2))
        .cache();

System.out.println("links.rdd().partitioner().toString():" + links.rdd().partitioner().toString())
System.out.println("links.toDebugString():" + links.toDebugString());
// Loads all URLs with other URL(s) link to from input file and initialize ranks of them to one.
JavaPairRDD<String, Double> ranks = links
        .mapValues(nbrs -> 1.0);

for (int current = 0; current < Integer.parseInt(args[1]); current++) {
    // Calculates URL contributions to the rank of other URLs.
    ranks = links.join(ranks).values()
            .flatMapToPair(joined -> {
                int urlCount = Iterables.size(joined._1);
                List<Tuple2<String, Double>> results = new ArrayList<Tuple2<String, Double>>();
                for (String n : joined._1) {
                    results.add(new Tuple2<String, Double>(n, joined._2() / urlCount));
                }
                return results.iterator();
            })
            .reduceByKey((a, b) -> a + b)
            .mapValues(sum -> 0.15 + sum * 0.85);
}
```