Word Count Example in Spark

The following slides show how to implement the MapReduce Word count algorithm, both in Java and Python

To understand the code, refer to

- The standard Java and Python APIs and reference manuals.
- Introduction to Programming in Spark (*for this course*)
- Spark RDD Programming guide.
- Spark Java API.
- Spark Python API.

All links and the complete code (`WordCountExample.Java` and `WordCountExample.py`) can be found in the course Moodle.

OUTLINE

1. Functional programming
2. 1-Round Word Count in Spark
3. 2-Round Word Count in Spark

Functional programming

Functional programming

One of the core ideas of functional programming is that functions can be arguments to other functions.

- In a MapReduce algorithm, the map and reduce functions used in each round can be regarded as "arguments" for the round.
- Spark's API relies heavily on passing functions. For example, many methods (e.g., RDD operations) require functions as arguments.

In Spark there are esentially 2 ways to pass functions as arguments to a method.

- as anonymous functions or lambdas which are defined inline without a name, where the actual argument is expected.
- as named functions defined outside the method.

Functional programming: Java example

Let `values` be a variable of type `JavaRDD<Double>`

USE of ANONYMOUS FUNCTIONS

denotes a generic element of the RDD values

- **Single statement**:

```
JavaRDD<Double> squaredValues = values.map((x) -> x*x);
```

- **Multiple statements**:

```
double fixed = 1.5;
JavaRDD<Double> normValues = values.map((x) -> {
  double diff = fixed - x;
  return diff;
});
```

Obs: variable fixed is "captured" by the anonymous function

Functional programming: Java example

USE of NAMED FUNCTIONS

Create a separate class (in a separate file) such as:

```
public class myOps {
  public static double mySquare(double x) {
    return x * x;
  }
}
```

Then, in the class containing the main program write:

```
JavaRDD<Double> squaredValues = values.map(myOps::mySquare);
```

Functional programming: Python example

Let `values` be an RDD of double

USE of ANONYMOUS FUNCTIONS

```
squaredValues = values.map(lambda x: x * x)
```

Obs: Python does not allow multiple statments in anonymous functions.

USE of NAMED FUNCTIONS

```
def mySquare(x):
        return x * x

squaredValues = values.map(mySquare)
```

1-Round Word Count in Spark

JAVA

Java: initialization

```
import ....

public class WordCountExample{

 public static void main(String[] args) throws IOException {

   // SPARK SETUP
   SparkConf conf = new SparkConf(true).setAppName("WordCount");
   JavaSparkContext sc = new JavaSparkContext(conf);

   // INPUT READING
   int K = Integer.parseInt(args[0]);
   JavaRDD<String> docs = sc.textFile(args[1]).repartition(K).cache();
```

CLI arguments

path to a file

Allows you to pass some configuration properties as command-line arguments

sc ≡ Spark context

repartition(K) ≡ performs a random partitioning of the elements of the RDD

cache() ≡ upon occurrence of the first action tries to store all data in main memories

docs ≡ RDD of strings where each element is a line of the input file, that is one of the input documents.

Java: implementation of the round

```
JavaPairRDD<String, Long> wordCounts = docs
    .flatMapToPair(f1) // <-- MAP PHASE
    .groupByKey()      // <-- SHUFFLING+GROUPING
    .mapValues(f2);    // <-- REDUCE PHASE
```

f1: transforms each document (a string of the RDD docs) into a set of (word, count) pairs

f2: for each (word, list(counts)) transforms the list of counts into their sum

Java: implementation of the round

```
JavaPairRDD<String, Long> wordCounts = docs
 .flatMapToPair((document) -> {    // <-- MAP PHASE
     String[] tokens = document.split(" ");
     HashMap<String, Long> counts = new HashMap<>();
     ArrayList<Tuple2<String, Long>> pairs = new ArrayList<>();
     for (String token : tokens) {
        counts.put(token, 1L + counts.getOrDefault(token, 0L));
     }
     for (Map.Entry<String, Long> e : counts.entrySet()) {
        pairs.add(new Tuple2<>(e.getKey(), e.getValue()));
     }
     return pairs.iterator();
 })
 .groupByKey()
 .mapValues((it) -> { // <-- REDUCE PHASE (R1)
     long sum = 0;
     for (long c : it) sum += c;
     return sum;
 })
```

f1

function f2

* flatMapToPair : transforms each input element into 0, 1 or more key value pairs.
If instead you want to implement a 1-1 transformation (1 element → 1 (key, value) pair)
then you can use mapToPair
The pairs returned by flatMapToPair are returned as an iterator

* mapValues : it computes the following transformation of every element resulting from groupByKey()

(word, listOfCounts) $\longrightarrow$ (word, $\sum_{c \in listOfCounts} c$)

f acts on

Java: implementation of the round

Alternatively, the combination of `groupByKey` and `mapValues` can be replaced by `reduceByKey` as follows:

```
JavaPairRDD<String, Long> wordCounts = docs
 .flatMapToPair((document) -> {    // <-- MAP PHASE
     String[] tokens = document.split(" ");
     HashMap<String, Long> counts = new HashMap<>();
     ArrayList<Tuple2<String, Long>> pairs = new ArrayList<>();
     for (String token : tokens) {
        counts.put(token, 1L + counts.getOrDefault(token, 0L));
     }
     for (Map.Entry<String, Long> e : counts.entrySet()) {
        pairs.add(new Tuple2<>(e.getKey(), e.getValue()));
     }
     return pairs.iterator();
 })
 .reduceByKey((x, y) -> x+y);    // <-- REDUCE PHASE
```

↳ substitutes groupByKey and mapValues

reduceByKey

* The argument is a function $D \times D \longrightarrow D$ where D is the domain of the values of the pairs
  The function must COMMUTATIVE and ASSOCIATIVE
* For each key k it combines the values of the pairs with key k as follows:

(k, V_1), (k, V_2) $\rightarrow (k, V_1+V_2)$
(k, V_3), (k, V_4) $\rightarrow (k, V_3+V_4)$
$\rightarrow (k, \Sigma V_i)$

The order in which the aggregation is executed is not specified but it EXPLOITS PARTITIONS

Java: implementation of the round

Also, one might define the function passed to `flatMapToPair` in a separate class `MyMethods` as follows:

```
public class MyMethods {
  public static Iterator<Tuple2<String, Long>>
                          processDoc(String document) {
    String[] tokens = document.split(" ");
    HashMap<String, Long> counts = new HashMap<>();
    ArrayList<Tuple2<String, Long>> pairs = new ArrayList<>();
    for (String token : tokens) {
        counts.put(token, 1L + counts.getOrDefault(token, 0L));
    }
    for (Map.Entry<String, Long> e : counts.entrySet()) {
        pairs.add(new Tuple2<>(e.getKey(), e.getValue()));
    }
    return pairs.iterator();
  }
}
```

Java: implementation of the round

and use function `processDoc` in class `WordCountExample` as follows:

```
JavaPairRDD<String, Long> wordCounts = docs
 .flatMapToPair(MyMethods::processDoc)  // <-- MAP PHASE
 .reduceByKey((x, y) -> x+y);           // <-- REDUCE PHASE
```

PYTHON

Python: main function

```
def main():

# SPARK SETUP
conf = SparkConf(true).setAppName('WordCount').setMaster("local[*]")
sc = SparkContext(conf=conf)

# INPUT READING
K = sys.argv[1]
K = int(K)
data_path = sys.argv[2]
docs = sc.textFile(data_path).repartition(K).cache()

    # COMPUTATION OF WORD COUNTS
wordCounts = word_count_1(docs)

if __name__ == "__main__":
main()
```

=> docs is an RDD of strings

RDD of (word, count) pairs

setMaster("local[*]") : configuration for local mode. When you want to execut the program on a cluster you will write yarn instead of local[*] (yarn = our cluster menager)

word_count_1 : function that defines the transformations applied to docs

Python: relevant functions

```
import ...

def word_count_per_doc(document):
    pairs_dict = {}
    for word in document.split(' '):
        if word not in pairs_dict.keys():
            pairs_dict[word] = 1
        else:
            pairs_dict[word] += 1
    return [(key, pairs_dict[key]) for key in pairs_dict.keys()]

def word_count_1(docs):
    word_count = (docs.flatMap(word_count_per_doc) # <-- MAP PHASE
        .groupByKey()                              # <-- SHUFFLE+GROUPING
        .mapValues(lambda vals: sum(vals))) # <-- REDUCE PHASE
    return word_count
```

dictionary to compute the counts within a document

→ RDD of (word, count) pairs

Python: relevant functions

Alternatively:

```
import ...

def word_count_per_doc(document):
    pairs_dict = {}
    for word in document.split(' '):
        if word not in pairs_dict.keys():
            pairs_dict[word] = 1
        else:
            pairs_dict[word] += 1
    return [(key, pairs_dict[key]) for key in pairs_dict.keys()]

def word_count_1(docs):
    word_count = (docs.flatMap(word_count_per_doc) # <-- MAP PHASE
        .reduceByKey(lambda x, y: x + y))          # <-- REDUCE PHASE
    return word_count
```

↳ replaces groupByKey + mapValues

* Java vs Python

  flatMapToPair → flatMap

  mapToPair → map

* word_count_per_doc : is applied to each document to produce (word, count) pairs (partial counts)

2-Round Word Count in Spark

2-Round Word Count in Spark

We will see 3 alternative implementations:

1. Implementation based on Improved word count assigning random keys *explicitly*
2. Implementation based on Improved word count assigning random keys *on the fly*
3. Implementation based on Spark partitions.

JAVA

Java: improved word count (explicit random keys)

```
JavaPairRDD<String, Long> wordCounts = docs
 .flatMapToPair((document) -> {    // <-- MAP PHASE (R1)
    String[] tokens = document.split(" ");
    HashMap<String, Long> counts = new HashMap<>();
    ArrayList<Tuple2<Integer, Tuple2<String, Long>>>
        pairs = new ArrayList<>();
    for (String token : tokens) {
        counts.put(token, 1L + counts.getOrDefault(token, 0L));
    }
    for (Map.Entry<String, Long> e : counts.entrySet()){
        pairs.add(new Tuple2<>(randomGenerator.nextInt(K),
                  new Tuple2<>(e.getKey(), e.getValue())));
    }
    return pairs.iterator();
 })
 .groupByKey()    // <-- SHUFFLE AND GROUPING
```

RDD of strings

random key

(word, count) pair

grouping based on random keys

```
.flatMapToPair((element) -> { // <-- REDUCE PHASE (R1)
   HashMap<String, Long> counts = new HashMap<>();
   for (Tuple2<String, Long> c : element._2()) {
       counts.put(c._1(), c._2() + counts.getOrDefault(c._1(), 0L));
   }
   ArrayList<Tuple2<String, Long>> pairs = new ArrayList<>();
   for (Map.Entry<String, Long> e : counts.entrySet()) {
       pairs.add(new Tuple2<>(e.getKey(), e.getValue()));
   }
   return pairs.iterator();
 })
.reduceByKey((x, y) -> x+y); // <-- REDUCE PHASE (R2)
```

First flatMapToPair

* Document → {(key, (word, count))} ← random key in [0, K-1]

groupByKey : creates (key, list of (word, count) pairs) pairs where keys are the random keys

Second flatMapToPair

* removes the random keys
* computes counts of each word limited to the occurrences in the group

reduceByKey : compute final counts

Java: improved word count (random keys on-the-fly)

```
JavaPairRDD<String, Long> wordCounts = docs
 .flatMapToPair((document) -> {    // <-- MAP PHASE (R1)
    String[] tokens = document.split(" ");
    HashMap<String, Long> counts = new HashMap<>();
    ArrayList<Tuple2<String, Long>> pairs = new ArrayList<>();
    for (String token : tokens) {
        counts.put(token, 1L + counts.getOrDefault(token, 0L));
    }
    for (Map.Entry<String, Long> e : counts.entrySet()) {
        pairs.add(new Tuple2<>(e.getKey(), e.getValue()));
    }
    return pairs.iterator();
 })
 .groupBy((wordcountpair) -> randomGenerator.nextInt(K))
 // IMPLEMENTS RANDOM KEY ASSIGNMENT, SHUFFLE AND GROUPING
```

As in the traditional alg.

assignment of random keys on the fly

(key, list of (word, count) pairs)
key is reduced

```
.flatMapToPair((element) -> { // <-- REDUCE PHASE (R1)
   //
   // AS BEFORE
   //
 })
.reduceByKey((x, y) -> x+y); // <-- REDUCE PHASE (R2)
```

groupBy(. . .)

* Assigns a random key to each (word, count) pair produced by the first flatMapToPair
* groups data into (key, list of (word, count) pairs) pairs, when key is random.

Java: use of Spark partitions

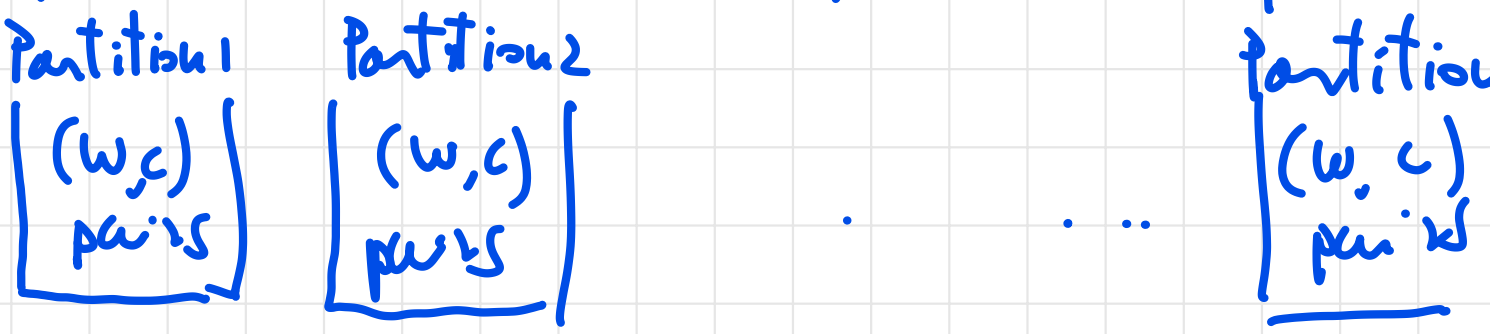
```
JavaPairRDD<String, Long> wordCounts = docs
 .flatMapToPair((document) -> {    // <-- MAP PHASE (R1)
    //
    // SAME AS IN THE 1-ROUND ALGORITHM
    //
 })
 .mapPartitionsToPair((element) -> {    // <-- REDUCE PHASE (R1)
    HashMap<String, Long> counts = new HashMap<>();
    while (element.hasNext()){
       Tuple2<String, Long> tuple = element.next();
       counts.put(tuple._1(), tuple._2() +
                              counts.getOrDefault(tuple._1(), 0L));
    }
    ArrayList<Tuple2<String, Long>> pairs = new ArrayList<>();
    for (Map.Entry<String, Long> e : counts.entrySet()) {
        pairs.add(new Tuple2<>(e.getKey(), e.getValue()));
    }
    return pairs.iterator();
 })
```

iterator of (word, count) pairs from one partition

(word, count)

```
.groupByKey()          // <-- SHUFFLE+GROUPING
.mapValues((it) -> { // <-- REDUCE PHASE (R2)
   long sum = 0;
   for (long c : it) sum += c;
   return sum;
 });
 // reduceByKey CAN BE USED IN PLACE OF groupByKey AND mapValues
```

mapPartitionsToPair exploits the Spark partitions



* processes each partition "element", that is a list of (word, count) pairs provided as an iterator
* creates (word, count) pairs when counts are aggregated at partition level

PYTHON

Python: relevant functions

```
def word_count_per_doc(document, K=-1):
  pairs_dict = {}
  for word in document.split(' '):
if word not in pairs_dict.keys():
        pairs_dict[word] = 1
else:
  pairs_dict[word] += 1
  if K == -1:
return [(key, pairs_dict[key]) for key in pairs_dict.keys()]
  else:
return [(rand.randint(0,K-1),(key, pairs_dict[key]))
                                  for key in pairs_dict.keys()]
```

K=-1 Doc → {(word, count)}

k≠-1 Doc → {(key, (word, count))}

random in [0, K-1]

Python: relevant functions

(key, list of (word, count) pairs)

```
def gather_pairs(pairs):
  pairs_dict = {}
  for p in pairs[1]:
    word, occurrences = p[0], p[1]
    if word not in pairs_dict.keys():
        pairs_dict[word] = occurrences
    else:
        pairs_dict[word] += occurrences
  return [(key, pairs_dict[key]) for key in pairs_dict.keys()]
```

aggregates counts for the same word

Python: improved word count (explicit random keys)

RDD of strings — element of docs

```
def word_count_2(docs, K):
  word_count = (docs.flatMap(lambda x: word_count_per_doc(x, K))
                                        # <-- MAP PHASE (R1)
    .groupByKey()                       # <-- SHUFFLE+GROUPING
    .flatMap(gather_pairs)              # <-- REDUCE PHASE (R1)
    .reduceByKey(lambda x, y: x + y)) # <-- REDUCE PHASE (R2)
  return word_count
```

After first flatMap: RDD whose elements are pairs (key, (word, count)) key $\in [0, K-1]$ random

After groupByKey: RDD whose elements are pairs (key, list of (word, count) pairs)

After second flatMap: RDD whose elements are pairs (word, count) (at most K pairs for every word)

After reduceByKey: RDD whose elements are pairs (word, count) 1 per word

Python: improved word count (random keys on-the-fly)

(word, count)

parameter K = -1

```
def word_count_3(docs, K):
  word_count = (docs.flatMap(word_count_per_doc) # <-- MAP PHASE (R1)
    .groupBy(lambda x: (rand.randint(0,K-1))) # <-- SHUFFLE+GROUPING
    .flatMap(gather_pairs)                     # <-- REDUCE PHASE (R1)
    .reduceByKey(lambda x, y: x + y))         # <-- REDUCE PHASE (R2)
  return word_count
```

as before

Python: use of Spark partitions

```
def gather_pairs_partitions(pairs):
  pairs_dict = {}
  for p in pairs:
    word, occurrences = p[0], p[1]
    if word not in pairs_dict.keys():
        pairs_dict[word] = occurrences
    else:
        pairs_dict[word] += occurrences
  return [(key, pairs_dict[key]) for key in pairs_dict.keys()]
```

```
def word_count_with_partition(docs):
  word_count = (docs.flatMap(word_count_per_doc) # <-- MAP PHASE (R1)
    .mapPartitions(gather_pairs_partitions)  # <-- REDUCE PHASE (R1)
    .groupByKey()                            # <-- SHUFFLE+GROUPING
    .mapValues(lambda vals: sum(vals))) # <-- REDUCE PHASE (R2)
  return word_count
```

k = -1

or alternatively, reduceByKey