Big Data
Hadoop

Dr. Wenceslao PALMA
wenceslao.palma@pucv.cl

Word Count Example

The goal of this example is to count the number of distinct words in a given text.

```
class MAPPER
  method MAP(docID a, doc d)
     for all term t in doc d do
       EMIT(term t, count 1)
```

The MAP method takes an input pair and produces a set of intermediate <key,value> pairs. Then all the intermediate values associated with the same intermediate key are grouped by the MapReduce library (shuffle phase).

Word Count Example

```
class REDUCER
  method REDUCE(term t, counts[c1,c2,...])
    sum = 0
    for all count c in counts[c1,c2,...] do
        sum = sum + c
    EMIT(term t, count sum)
```

The REDUCE method receives an intermediate key and a set of values for that key merging together these values to form a smaller set of values.

Word Count Example

Suposse we are give the following input file:

```
We are not what
we want to be,
but at least
we are not what
we used to be.
```

The MapReduce job consists of the following:

```
Map(doc_id, record) --> [(word, 1)]
Reduce(word, [1,1,...]) --> (word, count)
```

In the map phase the text is tokenized into words. Then a <word,1> pair is formed with these words.

```
<we, 1>; <are, 1>; <not, 1>; <what, 1>; ....
```

Remember that <key, value> pairs are generated in parallel on many machines. Each task has a little part of the overall Map input

Word Count Example

Considering our input text, in preparation for the reduce phase all the "we" pairs are grouped togheter, all the "what" pairs are grouped togheter, etc.

```
<we, 1> <we, 1> <we, 1> <we, 1> -- > <we, [1,1,1,1]>
<are, 1> <are, 1>               -- > <are, [1,1]>
<not, 1> <not, 1>               -- > <not, [1,1]>
...
```

In the reduce phase a reduce function is called once for each key. The reduce phase also sorts the output into increasing order by key as follows:

```
<are, 2>; <at, 1>; <be, 2>; <but, 1>; <least, 1>; <not, 2>; <to, 2>;
<used, 1>; <want, 1>; <we, 4>; <what, 2>
```

Like in the map phase, the reduce phase is also run in parallel. Each machine is assigned a subset of the keys to work on. The results are stored into a separate file.

Word Count::The Map source code

```
public class Map extends MapReduceBase implements Mapper<LongWritable, Text,
                                                          Text, IntWritable> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output,
                  Reporter reporter) throws IOException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      output.collect(word, one);
    }
  }
}
```

- LongWritable, Text, Text and IntWritable are Hadoop specific data types designed for operational efficiency. All these data types are based out of Java data types; LongWritable is the equivalent for long, IntWritable for int and Text for String.
- Mapper<LongWritable, Text, Text, IntWritable> refers to the data type of input and output key value pairs. The input key (LongWritable) is a default value, the input value (Text) is a line. The output is of the format <word,1> hence the data type of the ouput is Text and IntWritable.

Word Count::The Map source code

```
public class Map extends MapReduceBase implements Mapper<LongWritable, Text,
                                                          Text, IntWritable> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output,
                  Reporter reporter) throws IOException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      output.collect(word, one);
    }
  }
}
```

- In the map method **map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter)**
- The first two parameters refer to the data type of the input to the mapper.
- The third parameter **OutputCollector<Text, IntWritable> output** does the job of taking the output data from the mapper. The Reporter is used to report the task status internally in Hadoop environment.

Word Count::The Reduce source code

```
public class Reduce extends MapReduceBase implements Reducer<Text, IntWritable,
                                                              Text, IntWritable> {
 public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,
                    IntWritable> output, Reporter reporter) throws IOException {
  int sum = 0;
  while (values.hasNext()) {
    sum += values.next().get();
  }
  output.collect(key, new IntWritable(sum));
 }
}
```

- Considering Text, IntWritable, Text, IntWritable, the first two refers to data type of the input (<we,1>) to the reducer. The last two refers to data type of the output (<we,#occurrences>).
- In the reduce method **reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter)**
- The input to reduce method from the mapper after the sort and shuffle phase is of the format <we,[1,1,1,1]>

Word Count::The driver

```
public static void main(String[] args) throws Exception {
   JobConf conf = new JobConf(WordCount.class);
   conf.setJobName("wordcount");

   conf.setOutputKeyClass(Text.class);
   conf.setOutputValueClass(IntWritable.class);

   conf.setMapperClass(Map.class);
   conf.setCombinerClass(Reduce.class);
   conf.setReducerClass(Reduce.class);

   conf.setInputFormat(TextInputFormat.class);
   conf.setOutputFormat(TextOutputFormat.class);

   FileInputFormat.setInputPaths(conf, new Path(args[0]));
   FileOutputFormat.setOutputPath(conf, new Path(args[1]));

   JobClient.runJob(conf);
}
```

Compilation and run

```
$ mkdir classes
$ javac -classpath /usr/share/hadoop/hadoop-core-0.20.204.0.jar -d classes/ *.java
$ jar -cvf wordcount.jar -C classes/ .
$ hadoop dfs -ls input/
$ hadoop jar wordcount.jar org.myorg.WordCount input/ output/
$ hadoop dfs -cat output/part-00000
```

Word Count::python (mapper.py)

```
#!/usr/bin/env python

import sys

for line in sys.stdin:
    line = line.strip();
    words = line.split();
    for word in words:
        print '%s\t%s' % (word,1)
```

Word Count::python (reducer.py)

```
#!/usr/bin/env python
import sys

currentWord = None
wordCount = 0
word = None
for line in sys.stdin:
    line = line.strip();
    word,count = line.split('\t',1)
    count = int(count)

    if currentWord == word:
        wordCount += int(count)
    else:
       if currentWord:
          print '%s\t%s' % (currentWord,wordCount)
       wordCount = count
       currentWord = word
if currentWord == word:
   print '%s\t%s' % (currentWord,count)
```

Word Count::python

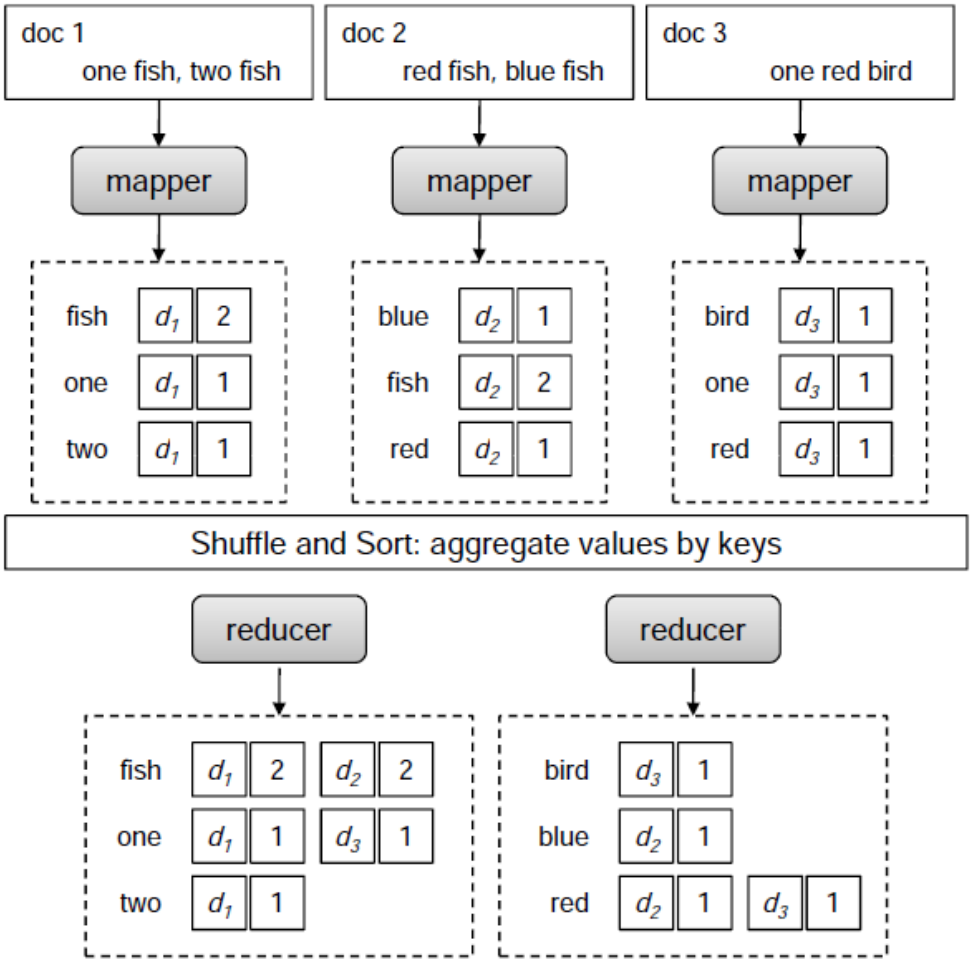
Testing locally before running in Hadoop

```
$ echo "hadoop linux hadoop big mapreduce hadoop" | python mapper.py | sort | python reducer.py
$ wget ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt
$ cat t8.shakespeare.txt | python mapper.py | sort | python reducer.py > output.txt
$ more output.txt
```

Word Count:: testing the code in a cluster

```
# copy the source code into the master node of the cluster
# replace joe with your username
$ scp sourceCodePython.zip joe@IPAddress:/home/joe

# login into the master node
$ ssh joe@IPAddress

# unzip the source code
$ unzip sourceCodePython.zip

# download the input data
$ wget ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt

# copy the input data into HDFS
$ hadoop fs -copyFromLocal t8.shakespeare.txt /user/joe/

# run the mapreduce code
$ hadoop jar /opt/cloudera/parcels/CDH-5.8.2-1.cdh5.8.2.p0.3/jars/\
  hadoop-streaming-2.6.0-cdh5.8.2.jar \
  -file /home/joe/mapper.py  -mapper /home/joe/mapper.py \
  -file /home/joe/reducer.py -reducer /home/joe/reducer.py \
  -input /user/joe/t8.shakespeare.txt -output /user/joe/output

# complete reference to hdfs commands
$ https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-common/FileSystemShell.ht
```

Exercise::Inverted index



Pseudocode

```
class MAPPER
  method MAP(docID n, doc d)
     H = new AssociativeArray

     for all term t in doc d do
       H{t} = H{t}+1

     for all term t in H do
       EMIT(term t, posting <n,H{t}>)


class REDUCER
  method REDUCE(term t, posting [<n1,f1>,<n2,f2>....])
     P = new List

     for all posting <docid,f> in postings [<n1,f1>,<n2,f2>....] do
         Append(P,<docid,f>)
     Sort(P)
     EMIT(term t, postings P)
```