Data Analytics Programming on Spark: Python In The Cloud

Chris Jermaine
cmj4@cs.rice.edu

By 2010, Hadoop Was Widely Used

- Open-source implementation of Google MapReduce
  - — Had Hadoop MapReduce
  - — Plus Hadoop distributed file system
- But the cracks were appearing in the edifice...

Hadoop MR Word Count Java Code

```
import java.util.*;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.IntWritable;

public class WordCount {

  public static int main(String[] args) throws Exception {

    // if we got the wrong number of args, then exit
    if (args.length != 4 || !args[0].equals ("-r")) {
      System.out.println("usage: WordCount -r <num reducers> <input> <output>");
      return -1;
    }

    // Get the default configuration object
    Configuration conf = new Configuration ();

    // now create the MapReduce job
    Job job = new Job (conf);
    job.setJobName ("WordCount");

    // we'll output text/int pairs (since we have words as keys and counts as values)
    job.setMapOutputKeyClass (Text.class);
    job.setMapOutputValueClass (IntWritable.class);

    // again we'll output text/int pairs (since we have words as keys and counts as values)
    job.setOutputKeyClass (Text.class);
    job.setOutputValueClass (IntWritable.class);

    // tell Hadoop the mapper and the reducer to use
    job.setMapperClass (WordCountMapper.class);
    job.setCombinerClass (WordCountReducer.class);
    job.setReducerClass (WordCountReducer.class);

    // we'll be reading in a text file, so we can use Hadoop's built-in TextInputFormat
    job.setInputFormatClass (TextInputFormat.class);

    // we can use Hadoop's built-in TextOutputFormat for writing out the output text file
    job.setOutputFormatClass (TextOutputFormat.class);

    // set the input and output paths
    TextInputFormat.setInputPaths (job, args[2]);
    TextOutputFormat.setOutputPath (job, new Path (args[3]));

    // set the number of reduce paths
    try {
      job.setNumReduceTasks (Integer.parseInt (args[1]));
    } catch (Exception e) {
      System.out.println("usage: WordCount -r <num reducers> <input> <output>");
      return -1;
    }

    // force the mappers to handle one megabyte of input data each
    TextInputFormat.setMinInputSplitSize (job, 1024 * 1024);
    TextInputFormat.setMaxInputSplitSize (job, 1024 * 1024);

    // this tells Hadoop to ship around the jar file containing "WordCount.class" to all of
the different
    // nodes so that they can run the job
    job.setJarByClass(WordCount.class);

    // submit the job and wait for it to complete!
    int exitCode = job.waitForCompletion (true) ? 0 : 1;
    return exitCode;
```

Not pretty!
Programmer burden too high...

Many Felt Hadoop MR Too Slow

- Data read from HDFS again for each MR job
- Bad for iterative data processing
  - — Many analytics tasks process and reprocess data

And API Too Restrictive

- Can only do Map
- Or MapReduce
- Everything else in terms of those operations... annoying!

So Hadoop MR Used Less and Less

- Issues led to other dataflow platforms to replace Hadoop MR
  - — Notable are Spark and Flink
  - — We will study Spark
- Interesting fact:
  - — While Hadoop MR is dying...
  - — Hadoop DFS is going strong
  - — De-facto standard for Big Data management
  - — Looks set to stay that way for a very long time

Apache Spark

- #1 Hadoop MR killer
- What is Spark?
  - — Platform for efficient, distributed data analytics
- Runs on the JVM
- Written in Scala
  - — But bindings for Java, Scala, Python, R
  - — Python nice for mathematical programming (but IPC overhead)
- Doesn’t do storage
  - — Focus exclusively on compute
  - — Commonly used with HDFS, S3, HBase, etc.

RDDs

- Basic abstraction: **Resilient Distributed Data Set** (RDD)
- RDD is a data set buffered in RAM by Spark
  - — Distributed across machines in cluster
  - — To create and load an RDD (in Python):

```
myRDD = sc.textFile (someFileName) # sc is the Spark context
# or else...
data = [1, 2, 3, 4, 5]
myRDD = sc.parallelize (data) # or
myRDD = sc.parallelize (range (20000)) # or...
```

Computations: Series of Xforms Over RDDs

- Example: word count
  - — Count number of occurs of each distinct word in a corpus

```
def countWords (fileName):
     textFile = sc.textFile (fileName)
     lines = textFile.flatMap (lambda line: line.split(“ “))
     counts = lines.map (lambda word: (word, 1))
     aggCounts = counts.reduceByKey (lambda a, b: a + b)
     retrun aggCounts.top (200, key=lamda p: p[1])
```

- What transforms do we see here?
  - — `flatMap`, `map`, `reduceByKey`, `top`
- Let’s go through them...

But First… What's a Lambda?

- Basically, a function that I can pass like a variable
- Key ability: can "capture" its surroundings at creation

```
def addTwelveToResult (myLambda):
     return myLambda (3) + 12

a = 23
aCoolLambda = lambda x : x + a
addTwelveToResult (aCoolLambda) # prints 48

a = 45
addTwelveToResult (aCoolLambda) # prints ???
```

Lambdas and Comprehensions

- Lambdas can return many items

```
def sumThem (myLambda):
     tot = 0
     for a in myLambda ():
          tot = tot + a
     return tot

x = np.array([1, 2, 3, 4, 5])
iter = lambda : (j for j in x)
sumThem (iter) # prints 15
```

Lambdas and Comprehensions

- Lambdas can return many items

```
def sumThem (myLambda):
     tot = 0
     for a in myLambda ():
          tot = tot + a
     return tot

x = np.array([1, 2, 3, 4, 5])
iter = lambda : (j for j in x)
sumThem (iter) # prints 15
```

- Exercise: change `iter` so accepts param, adds to each item, then:

```
def sumThem (myLambda, addMeIn):
     tot = 0
     for a in myLambda (addMeIn):
          tot = tot + a
     return tot                  # try sumThem (iter, 5)
```

Basic Spark Operations

flatMap ()

```
def countWords (fileName):
     textFile = sc.textFile (fileName)
     lines = textFile.flatMap (lambda line: line.split(“ “))
```

- Process every data item in the RDD
- Apply lambda to it
- Lambda argument to return zero or more results
- Each result added to resulting RDD

map ()

```
def countWords (fileName):
     textFile = sc.textFile (fileName)
     lines = textFile.flatMap (lambda line: line.split(“ “))
     counts = lines.map (lambda word: (word, 1))
```

- Process every data item in the RDD
- Apply lambda to it
- But the lambda must return **exactly** one result

reduceByKey ()

```
def countWords (fileName):
     textFile = sc.textFile (fileName)
     lines = textFile.flatMap (lambda line: line.split(“ “))
     counts = lines.map (lambda word: (word, 1))
     aggCounts = counts.reduceByKey (lambda a, b: a + b)
```

- Data must be (*Key*, *Value*) pairs
- Shuffle so that all (*K*, *V*) pairs with same *K* on same machine
- Organize into (*K*, (V_1, V_2, ..., V_n)) pairs
- Use the lambda to “reduce” the list to a single value

top ()

```
def countWords (fileName):
     textFile = sc.textFile (fileName)
     lines = textFile.flatMap (lambda line: line.split(“ “))
     counts = lines.map (lambda word: (word, 1))
     aggCounts = counts.reduceByKey (lambda a, b: a + b)
     retrun aggCounts.top (200, key=lamda p: p[1])
```

- Data must be (*Key, Value*) pairs
- Takes two params... first is number to return
- Second (optional): lambda to use to obtain key for comparison
- Note: `top` **collects** an RDD, moving from cloud to local
- So result is **not** an RDD

filter ()

- Not used in example code
- But needed for first activity
- Accepts a boolean-valued lambda
- That lambda applied to each item in RDD
- Item removed from result if and only if lambda returns `false`

An Important Note

- Lazy evaluation... if I run this code:

```
textFile = sc.textFile (fileName)
lines = textFile.flatMap (lambda line: line.split(“ “))
counts = lines.map (lambda word: (word, 1))
aggCounts = counts.reduceByKey (lambda a, b: a + b)
```

- Nothing happens! (Other than Spark remembers the ops)
  - — Spark does not execute until an attempt made to collect an RDD
  - — When we hit `top()`, **then** all of these are executed
- Why do this?
  - — By waiting until last possible second, opportunities for “pipelining” exploited
  - — Only ops that require a shuffle can’t be pipelined

Another Important Note

- Lambda capture **by value** at a difficult-to-determine instant
  - — All referenced variables serialized and broadcast
  - — Sometime between definition of RDD transform...
  - — And lazy evaluation
  - — Very useful! Standard way of broadcasting local state to distributed computations
  - — So be careful! Never rely on side-effects...

Some Other, More Advanced Ops

— `join ()`, `groupByKey ()`, `aggregateByKey ()`

join ()

- Given two data sets `rddOne`, `rddTwo` of (*Key*, *Value*) pairs
- `rddOne.join (rddTwo)` returns (*K*, (*V1*, *V2*)) pairs
- constructed from all (*K1*, *V1*) from `rddOne`, (*K2*, *V2*) from `rddTwo`, where *K1 == K2*
- Can blow up RDD size if join is many-to-many
- Requires expensive shuffle!

groupByKey ()

- Data must be (*Key*, *Value*) pairs
- Shuffle so that all (*K*, *V*) pairs with same *K* on same machine
- Organize into (*K*, (V_1, V_2, ..., V_n)) pairs
- Store each list as a `ResultIterable` for future processing
- Like `reduceByKey ()` but without the reduce

aggregateByKey ()

- Like `reduceByKey ()`
- Data must be (*Key*, *Value*) pairs
- Organize into (K, (V_1, V_2, ..., V_n)) pairs
- Then aggregate the list, like `reduceByKey ()`
- With `reduceByKey ()` aggregate directly, can be restrictive...
  - — what if values are cities, and we want the list of unique cities?
- `aggregateByKey ()` takes three args
  - — The “zero” to init the aggregation
  - — Lambda that takes `x1`, `x2` and aggs them, where `x1` already aggregated, `x2` not
  - — Lambda that takes `x1`, `x2` and aggs them, where both already aggregated

Questions?