Hadoop Tutorial
Concurrency en Parallel Programmeren 2015,
University of Amsterdam

Contents

1 Overview

Hadoop MapReduce is a framework for writing applications which can process large volumes of data ("Big Data"), by doing the work in a distributed, parallel way.

As you learnt in the lecture, MapReduce tasks are split into two different parts; a *mapper*, which processes the input files in chunks and produces an intermediate output version, and a *reducer*, which takes sorted/merged versions of the mapper output as input, and outputs the final output files. These input and output files are stored on HDFS, the Hadoop Distributed Filesystem.

First, you need to install Hadoop, and make sure that you can access the Hadoop cluster we'll be using to run jobs. Then, we'll try building and running an example Hadoop application. Finally, we'll go through the application's code, which you'll use as a framework for your Hadoop assignment. Once you're done, you can start on the assignment.

Before we begin, a warning: This tutorial was rewritten at the last minute and may contain mistakes. Ask one of the assistants if you encounter any problems!

2 Installing Hadoop

In order to do the assignment, you'll need to be able to execute MapReduce jobs on the SURFsara Hadoop cluster, which we're going to be using this year, since DAS-4 has very limited capacity for running Hadoop jobs. SURFsara have kindly allowed our students to use this cluster for this assignment, but we can't guarantee it will always have enough capacity for your jobs; make sure you don't wait until the last day before the deadline to run your jobs. Also, please be **careful** and don't use their machines for anything other than submitting your Hadoop jobs and collecting your results.

We've assigned everyone accounts with the same name as their DAS-4 accounts (usernames '`cpp1501`', '`cpp1502`', etc), but the passwords are different. If you didn't get your password already, then ask us in the first computer lab, or via e-mail.

Jeroen Schot (from SURFsara) has put together a VirtualBox VM which you should be able to use, if you have difficulties making this work on your computer. You can find it at:
`http://beehub.nl/surfsara-hadoop/public/hathi-cpp2015.ova`

Read the rest of this section **carefully**, and make sure you follow **all** the instructions.

Before you can begin, you should download Hadoop by running this command:
`git clone https://github.com/sara-nl/hathi-client`

You must use this repository! Once you've downloaded this, you'll need to edit the `hathi-client/conf/settings.linux` script and add the path to your Java installation directory. If you're using Ubuntu, you can just uncomment the line under 'Java home for Ubuntu', and comment out the one under 'Java home for CentOS'.

Then, run '`cd hathi-client`' and '`source conf/settings.linux`' to add Hadoop to your path. You must do this in **every shell** before you can use Hadoop (or Kerberos) from it.

In order to submit jobs, you'll need a Kerberos ticket, which authenticates you to the system. You can get one by running '`kinit cpp1509@CUA.SURFSARA.NL`' (replacing `cpp1509` with your account name), and typing your password. If you don't have the kinit command on your system, it's part of Kerberos 5; you can install it with `apt-get install krb5-user` on Ubuntu/Debian. (Please don't log into the cluster using ssh unless you have no other way to work. The login machine is not allocated enough resources for us to use it.)

To make sure that you're able to run Hadoop, you can run the following command:

```
$ hadoop version
```

The output should look something like this:

```
Hadoop 2.6.0
Subversion https://git-wip-us.apache.org/repos/asf/hadoop.git -r e3496...
Compiled by jenkins on 2014-11-13T21:10Z
```

To make sure that you can communicate with the Hadoop cluster, you can try listing the running jobs on the cluster:

```
mapred job -list
```

The output should contain a list of jobs (although it might be empty). If you don't have your settings configured correctly, or you don't have a valid Kerberos ticket, you'll get an error message instead.

3 Running the WordCount Tutorial

We're going to start by running the standard WordCount example [1], which simply counts the occurrences of each word in a given input file. Make sure you do this! After we're done, you'll implement the actual assignment by continuing to build on top of this example.

3.1 Download the files

You should start by downloading the project code from Blackboard. It should contain a `pom.xml` file, which is a project file for the tool called Maven which you'll be using to build your code. It should also contain a `src` directory, which contains the source code (in a subdirectory called `main/java/nl/uva/cpp/`).

You also need an input file in which to count words; why not use the data file which you'll later be using? You can find it in `tweets2009-06-brg.txt`, which should also be available on Blackboard.

3.2 Building the code

You can install Maven on Ubuntu/Debian by running `apt-get install maven`. [2]

To compile your code and produce a `jar` file, automatically downloading any needed dependencies, run this:

```
$ mvn package
```

The project depends on the Hadoop API (`http://hadoop.apache.org/`), the Stanford Natural Language Processing API (`http://nlp.stanford.edu/`) and the JLangDetect API (`https://github.com/melix/jlangdetect`). These last two APIs will be used for sentiment analysis[3] and language identification[4].

After compiling, the `target` directory should (hopefully) contain a jar file called `wordcount-example-0.1-SNAPSHOT.jar`, which contains all the needed libraries, and the compiled code for your MapReduce job.

3.3 Running the code

You can run the code in two different ways:

1. Locally: This mode doesn't use the Hadoop cluster at all. It runs the code on a single machine e.g. your laptop/workstation and uses the local file system. You can use this mode for testing and debugging your code.

2. Remotely: This mode uses the Hadoop cluster to run your tasks, and HDFS for storage. You won't be able to see debug messages from your code, so you should only use this when running your final experiments.

For our assignment, we'll be using `tweets2009-06-brg.txt` as the input file, and an arbitrary directory (we suggest `output/`) as the output file. Remember, our tool expects these to be provived on the command line.

[1] This tutorial is based on the Hadoop tutorial; thanks to Jeroen Schot for helping adapt it.

[2] If you have problems with Maven, you can manually download the libraries you need, put them in a `lib` directory, and then compile the Java manually. We don't really support this method; please use Maven if you can.

[3] Sentiment analysis aims to determine the attitude of a person

[4] Detecting whether the written language is Dutch, English, etc.

3.3.1 Running the code locally

As long you have the input file on your local disk, you can run the job locally by typing:

```
$ mvn exec:java -Dexec.args="tweets2009-06-brg.txt output/ 1"
```

Or alternatively, if you're building it without Maven:

```
$ java -jar nl.uva.cpp.WordCount.jar tweets2009-06-brg.txt output/ 1
```

3.3.2 Running the code remotely

Before you run the job remotely on the Hadoop cluster, you have to copy the `tweets2009-06-brg.txt` to the HDFS. You can do this using the `hdfs` command:

```
$ hdfs dfs -copyFromLocal tweets2009-06-brg.txt
```

This will copy the `tweets2009-06-brg.txt` file to the cluster's HDFS, and store it inside your home directory.

To run the job remotely (in a fully-distributed mode, on the cluster), just run it using Hadoop (the 1 means that you want only one task, as you'll see later):

```
$ yarn jar target/wordcount-example-0.1-SNAPSHOT.jar nl.uva.cpp.WordCount
    tweets2009-06-brg1of2.txt output/ 1
```

Attention: If you attempt to run the job again, it won't work, because the output directory already exists. Instead, you'll get an error like:

```
Output directory hdfs://hathi-surfsara/user/cpp1509/output already exists
```

If you want to run the job again, you have to either delete the output folder (by running `hdfs dfs -rm -r output`), or specify a different one.

3.4 Monitoring

After you've submitted a job, the Hadoop client will wait until the job is finished (which might take some time, if the cluster is busy), and report progress. You'll see output like this, telling you that your job has been submitted:

```
15/11/17 16:28:30 INFO impl.YarnClientImpl: Submitted application
    application_1446724277285_9813
15/11/17 16:28:30 INFO mapreduce.Job: The url to track the job: http://head05.
    hathi.surfsara.nl:8088/proxy/application_1446724277285_9813/
15/11/17 16:28:30 INFO mapreduce.Job: Running job: job_1446724277285_9813
```

Do **not** try to use the provided URL (you have to authenticate using your Kerberos credentials, which is problematic). You can monitor the progress of your job using the command line like this:

```
$ mapred job -status job_1446724277285_9813
```

As we discussed earlier, you can also list all jobs which are running on the cluster:

```
$ mapred job -list
```

And you can kill your own jobs if necessary:

```
$ mapred job -kill <JOB ID>
```

3.5 Downloading the results

Finally, you can download the results from the remote HDFS by using the `-copyToLocal` parameter:

```
$ hdfs dfs -copyToLocal output
```

The output will be files with names starting in `part-`, in this directory. Take a look!

4 A closer look at the WordCount Tutorial

In this section, we'll be looking at the Java source code in the `src/main/java/nl/uva/cpp/` directory.

4.1 How the job works

When we run our application, the main function in `WordCount.java` is run, which simply starts the Hadoop `ToolRunner` with our tool.

The implementation of the tool itself is in `WordCountTool.java`. It does several important things, all of which you should make sure you look at:

- First, it sets the input and output task paths, which are provided on the command line. **Attention**: Remember, the input and output paths can be either local (if we are running the job in local mode), or HDFS paths (if we are running in remote/fully-distributed mode).
- Then, it sets the mapper/reducer classes to use, to the WordCountMapper and WordCountReducer tools which are also part of our application.
- We also need to set the input and output type of the input files for the mappers, so it does that. The input type is very important, because it specifies the way the input is going to be split. The `TextInputFormat` is meant for plain text files – files are broken into lines. Keys are the position in the file, and values are the line of text.
- Sets the number of tasks to run, which are also provided on the command line. We want to use the same number of mapper tasks as the number of reduce tasks, for simplicity. Since our input files are fairly small, by default, MapReduce will refuse to split the input lines into multiple pieces, which means we can only use a single mapper. We set the split size (in bytes) to a lower value, to force it to split the input into multiple pieces, so that we can get a useful number of mappers.
- Finally, it actually runs the job.

4.2 Mapper

The mapper is implemented in `WordCountMapper.java`. It outputs a key/value pair of (word, 1) for every word it sees in the input.

It is called once for every key in the input (in our case, this means that it's called for every line). It processes this input, and produces key/value pairs as output (which are then provided as input to the Reduce tasks).

In the provided example, we're just counting words – so the key we write is the word, and the value is always one.

In the method, we split each incoming line into words:

```
String line = value.toString().toLowerCase();
StringTokenizer itr = new StringTokenizer(line);
```

Then, we simply iterate through the words and emit them to the reducer:

```
    while (itr.hasMoreTokens()) {
      String token = itr.nextToken();
      word.set(token);
      context.write(word, one);
    }
```

The remaining line just increments a counter, as an example of how a mapper can report progress:

```
context.getCounter(Counters.INPUT_WORDS).increment(1);
```

4.3 Reduce

The `Reduce`'s task is simple: it just has to add the values (counts) of the keys (words) which have been emitted from the `Map` class:

```
  int sum = 0;
  int count = 0;
  for (IntWritable val : values) {
    sum += val.get();
    count++;
  }
```

After the `Reduce` class has added all these values, it writes the result as the output:

```
output.collect(key, new IntWritable(sum));
```

Alternatively, you could write some freeform (string) data to the output. To do this, you'd need to change the last `IntWritable` to `Text` in the `Reduce` class's `extends` clause, and the OutputValueClass in the WordCountTool `run` function (you'd also need to call `setMapOutputValueClass`, because your `Mapper` would still produce integers). Then, as an example:

```
String value = String.valueOf(count) + "\t" + String.valueOf(sum);
context.write(key, new Text(value));
```

As you saw earlier, the output of the `Reduce` class is saved in the output path of the job, in parts.

You should now be ready to begin the assignment.