Hadoop Tutorial
University of Amsterdam BSc Informatica
Concurrency & Parallel Programming

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 SURF- sara 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 it is a shared resource; make sure you don't wait until the last day before the deadline to run your jobs. 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 'cpp1701', 'cpp1702', 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.

There are two options, install the hadoop client on your own machine or use the login node that surf has provided for you at

```
login.hathi.surfsara.nl
```

The following instructions are if you want to run hadoop localy on your system:

Before you can begin, you must download Hadoop and the cluster configuration by running this command:

```
git clone https://github.com/sara-nl/hathi-client
```

and follow the instructions written in the readme file.

To move files to the $JAVA_HOME dir you will need to use sudo.

3 WordCount Tutorial

We're going to start by running the standard WordCount example 2, 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; you can just use one of the data files you'll have to use for the assignment. You can find it in `tweets2009-06-brg.txt` , which should also be available on Blackboard (it might have a different name, be warned).

3.2 Building the code

You can install Maven on Ubuntu/Debian by running `apt-get install maven`. To compile your code and produce a jar file, automatically downloading any needed depen- dencies, run this:

```
$ mvn package
```

You will need to run this every time you change your code. After compiling, the target directory should (hopefully) contain a jar file called `wordcount-example-0.1-SNAPSHOT.jar`, which contains the compiled code for your MapReduce job, and all the needed libraries. 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 analysis4 and language identification.

3.3 Running the code

You can run the code in two different ways:

1. Locally: This mode is not using 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.

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"
```

3.3.2 Running the code remotely

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-brg.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/<username>/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:

```
$ yarn application -status job_1446724277285_9813
```

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

```
$ yarn application -list
```

And you can kill your own jobs if necessary:

```
$ yarn application -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
```

If you want to merge all the output files to one large file you can use the following command:

```
    hadoop fs -getmerge -nl output results.txt
```

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/n-l/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:

1. 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).

2. Then, it sets the mapper/reducer classes to use, to the WordCountMapper and WordCountReducer tools which are also part of our application.

3. 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.

4. 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.

5. 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.