Polytechnique Montreal
LOG8415: Lab 2
MapReduce with Hadoop on AWS

Abstract

In this lab assignment, you will get hands-on experience running MapReduce on Hadoop and Spark using AWS. You will also learn how to fit problems/algorithms into the MapReduce paradigm. In the first part of the assignment, you will have to run a simple wordcount code example provided in the Hadoop tutorial. Next, you will apply your newly acquired skills to answer few business intelligence questions using Spark and MapReduce paradigm. You are asked to report your results and analysis by producing a report using LaTeX format.

Objectives

The overall goals of this lab assignment are:

- Get hands-on experience running MapReduce on AWS.
- Understand two famous big data tools: Hadoop and Spark.
- Run big data operations on Hadoop and Spark.
- Use the codebase you've developed for your 1st assignment to setup your instances.
- Learn how to fit problems/algorithms into the MapReduce paradigm.

Instructions on setting up Hadoop

You will find detailed instructions about Hadoop installation at the following website: Getting Hadoop up and running on Ubuntu [4]. Remember that by default, Hadoop is configured to run in a non-distributed or standalone mode.

Experiments with WordCount

Running WordCount.java on HDFS

Copying Data to HDFS

In order to process a text file with Hadoop, you first need to download the file to a directory in your Hadoop account, then copy it to the Hadoop Distributed File System (HDFS) so that the Hadoop Name Node and Data Nodes can share it.

Download a copy of James Joyce's Ulysses book available at [3] Create an "input" (or whatever your identifier is) directory in the Hadoop Distributed File System (HDFS) and copy the data file pg4300.txt to it:

hdfs dfs -mkdir input
(makes an input directory in the cloud)

hdfs dfs -copyFromLocal pg4300.txt input

WordCount.java MapReduce Program

Hadoop comes with a set of demonstration programs.

They are in /hadoop/src/examples/org/apache/hadoop/examples/. One of them is WordCount.java which will automatically compute the word frequency of all text files found in the HDFS directory you ask it to process. The program has several sections:

The map section:

```
public static class MapClass 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 itr = new StringTokenizer(line);
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      output.collect(word, one);
    }
  }
}
```

The Map class takes lines of text that are fed to it (the text files are automatically broken down into lines by Hadoop– No need for you to do it!) and breaks them into words. Outputs a datagram for each word that is a (String, int) tuple, of the form (“some-word", 1), since each tuple corresponds to the first occurrence of each word, so the initial frequency for each word is 1.

The next part is the Reducer which, reduces a set of intermediate values that share a key, to a smaller set of values. Reducer implementations can access the Configuration of the job via the JobContext.getConfiguration() method.

```
public static 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));
   }
 }
```

The reduce section gets collections of datagrams of the form [(word, n1), (word, n2)...] where all the words are the

same, but with different numbers. These collections are the result of a sorting process that is integrated in Hadoop, and which gathers all the datagrams with the same word together. The reduce process gathers the datagrams inside a Datanode and gathers datagrams from the different Datanodes into a final collection of datagrams, where all the words are now unique, with their total frequency (number of occurrences). The mapreduce organization section

```
conf.setMapperClass(MapClass.class);
 conf.setCombinerClass(Reduce.class);
conf.setReducerClass(Reduce.class);
```

Here we see that the combining stage and the reduce stage are implemented by the same reduce class, which makes sense, since the number of occurrences of a word as generated on several Datanodes is just the sum of the numbers of occurrences.

The datagram definitions

```
% the keys are words (strings) conf.setOutputKeyClass(Text.class);
% the values are counts (ints) conf.setOutputValueClass(IntWritable.class);
```

As the documentation indicates, the datagrams are of the form (String, int).

Running WordCount

Run the wordcount.java program from the example directory in Hadoop:

```
hadoop jar hadoop -0.19.2-examples.jar wordcount input output
```

Getting the output

1. Take a look at the output:
2. hdfs dfs -ls /output
3. The output folder will be created automatically. You just need to verify that a new directory with -output at the end of your identifier has been created. Then look at the content of this directory for results. hdfs dfs -ls output
4. Experiment: Run WordCount.java on Ulysses from your own HDFS directory, using Hadoop and measure the execution time.

Compare Hadoop vs. Linux

In this section, you will compare the performance of Hadoop with a regular PC running Linux and computing the word frequencies of the contents of Ulysses.

Step 1: time the execution of WordCount.java on Hadoop.

```
hdfs dfs -rm output
time hadoop jar hadoop-0.19.2-examples.jar wordcount input output
```

Observe and record the total execution time (real)

Step 3: compute the word frequency of a text with Linux, using Linux commands and pipes, as follows:
cat 4300.txt | tr ’␣’ ’␣[ret]␣’ | sort | uniq -c [ret]

where [ret] indicates that you should press the return/enter key. Try the command and verify that you get the word frequency of pg4300.txt.

Apache Spark

Apache Spark [1]: In contrast to Hadoop’s two-stage disk-based MapReduce paradigm, Spark’s multi-stage in-memory primitives provides performance up to 100 times faster for certain applications. You can begin by reading the official documentations provided here: http://spark.apache.org/docs/latest/

Setup

Create one M4.large Linux Ubuntu instance and set up Hadoop (latest version) and Spark (v2.0.0) on them. Make sure that all required packages are installed. You can use Java or Python (PySpark).

Target Datasets

Use below datasets for all your experiments:

https://tinyurl.com/4vxdw3pa
https://tinyurl.com/kh9excea
https://tinyurl.com/dybs9bnk
https://tinyurl.com/datumz6m
https://tinyurl.com/j4j4xdw6
https://tinyurl.com/ym8s5fm4
https://tinyurl.com/2h6a75nk
https://tinyurl.com/vwvram8
https://tinyurl.com/weh83uyn

- Method: Run WordCount program both on Hadoop and Spark machines 3 times on all above datasets and measure the execution time, plot all your data points and compare/report the average. You can launch the program in Hadoop by giving the JAR file as an argument and submit a work on Spark using the built-in bash script.

Performance Comparison

Apache Spark processes data in-memory while Hadoop MapReduce persists back to the disk after a map or reduce action, so Spark should outperform Hadoop MapReduce. We will investigate this hypothesis in our assignment.

-Objective: Run WordCount program in both Hadoop and Spark and compare the execution time.

-Guide: Spark examples [2]

Problem

Write a MapReduce program in Hadoop that implements a simple “People You Might Know" social network friendship recommendation algorithm. The key idea is that if two people have a lot of mutual friends, then the system should recommend that they connect with each other.

Instructions

Input Data:

Download the input file from Moodle. The input file contains the adjacency list and has multiple lines in the following format:

<User><TAB><Friends>
Here, <User> is a unique integer ID corresponding to a unique user and <Friends> is a comma separated list of unique IDs corresponding to the friends of the user with the unique ID <User>. Note that the friendships are mutual (i.e., edges are undirected): if A is friend with B then B is also friend with A. The data provided is consistent with that rule as there is an explicit entry for each side of each edge.

Algorithm

For each user U, recommends N = 10 users who are not already friends with U, but have the most number of mutual friends in common with U.

Output

The output should contain one line per user in the following format:

<User><TAB><Recommendations>
where <User> is a unique ID corresponding to a user and <Recommendations> is a comma separated list of unique IDs corresponding to the algorithm's recommendation of people that <User> might know, ordered in decreasing number of mutual friends. Even if a user has less than 10 second-degree friends, output all of them in decreasing order of the number of mutual friends. If a user has no friends, you can provide an empty list of recommendations. If there are recommended users with the same number of mutual friends, then output those user IDs in numerically ascending order.

Working with Groups

You should work in groups and submit only one report along with all necessary code.

Bonus Point

Using Microsoft Azure instead of AWS. If you go the Azure root, there is no need for automating your solution.

Report

One submission per group is required for this assignment. To present your findings and answers questions, you have to write a lab report on your results using LaTeX template. In your report should write:

- Experiments with WordCount program.
- Performance comparison of Hadoop vs. Linux.
- Performance comparison of Hadoop vs. Spark on AWS.
- Describe how you have used MapReduce jobs to solve the social network problem.
- Describe your algorithm to tackle the social network problem.

- Presents your recommendations of connection for the users with following user IDs: 924, 8941, 8942, 9019, 9020, 9021, 9022, 9990, 9992, 9993.
- Summary of results and instructions to run your code.

Evaluation

A single final submission for this assignment is due on the date specified in Moodle for each group. You need to submit one PDF file per group. All necessary codes, scripts and programs must be sufficiently commented and attached to your submission. A demo will be organized after the submission for your work. All team members should be participating in the demo.

Your assignment will be graded on presentation and content as follows:

2 pts: Hadoop and Spark installed, experiments done successfully.

3 pts: Comparing performance of Hadoop vs. Spark on AWS;

7 pts: Solving the social networking problem;

2 pts: Source code, scripts and results;

1 pts: General presentation and the quality of the report;

5 pts: Demo;

Please submit your PDF report and push your code to a GitHub repository.

Acknowledgement

We would like to thank Amazon Web Services for supporting us through their and AWS Educate grants.

References

1. [1] Apache spark. http://spark.apache.org. Accessed: 2021-10-01.
2. [2] Apache spark examples. http://spark.apache.org/examples.html. Ac-
3. [3] Gutenberg textual data. http://www.gutenberg.org/cache/epub/4300/pg4300.txt.
4. [4] Hadoop tutorial. http://dzone.com/articles/getting-hadoop-and-running.
5. [5] Mapreduce tutorial. https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.