Electrical and Computer Engineering / Software Engineering

CPrE/SE 419: Software Tools for Large-Scale Data Analysis, Spring 2023

Purpose

The main objective of this lab is to start your familiarizing with the MapReduce programming model under Hadoop. As discussed in class, MapReduce is an abstraction for processing large data sets and has been applied successfully to a number of tasks in the past, including web search. At the end of this lab, you will be able to:

- Write and Execute a program using Hadoop MapReduce
- Write algorithms for data processing using MapReduce
- Apply these skills in analyzing basic statistics of a large text corpus

Submission

Create a zip (or tar) archive with the following and hand it in through Canvas.

- A write-up report includes
  - Answering questions for each experiment in the lab, if there are any
  - Output for each experiment (screenshots)
- Commented Code for your program.
  - Include all source files needed for compilation
  - Name each source file according to the experiment, as appropriate

MapReduce

MapReduce is a programming model for parallel programming on a cluster. We will be working with the Hadoop implementation of the MapReduce paradigm.

A Program based on the MapReduce model can have several rounds. Each round must have a *map* and a *reduce* method. The input to the program is processed by the map method and it emits a sequence of key and value pairs <k, v>. The system groups all values corresponding to a given key and sorts them. Thus for each key k, the system generates a list of values corresponding to k, and this is then passed on to the reduce method. The output from the reduce methods can then be used as the input to the next round, or can be output to the distributed file system.

A key point is that *two (or more) mappers cannot communicate with each other*, and *neither can two (or more) reducers communicate with each other*. Although this restricts what a program can do, it allows for the system to easily parallelize the actions of the mappers and the reducers. The only communication occurs when the output of a mapper is sent to the reducers.

Resources

Take a look at the Example Hadoop Program found on Canvas **– "WordCount.java"**. A MapReduce program typically has at least 3 classes as shown in the example code: A **Driver Class** (in our case, WordCount)**,** a **Mapper Class** and a **Reducer Class.**

The method of compiling java program is same as Lab 1. However, besides of the properties and dependencies mentioned in Lab 1 instruction, there is one **more** dependency need to be added in **dependencies** section in pom.xml, shown as below

```
<dependency>

        <groupId>org.apache.hadoop</groupId>

        <artifactId>hadoop-mapreduce-client-core</artifactId>

        <version>3.1.3</version>

        <scope>provided</scope>

</dependency>
```

Don't forget to change java version to 1.8 and Don't forget to start-all via terminal shell script

Notes

- The MapReduce programs read input from the HDFS and write their output to HDFS.

- Two Hadoop programs running simultaneously cannot have the same output path, although they can share the same input path. Thus, make your output path unique, as otherwise your job may have a conflict with the output of other jobs, and hence fail. Finally, make sure that the output directory is empty by deleting its contents. You must do this check every time you re-run the program or it will fail.

- Note that each MapReduce round can have multiple input paths, but only one output path assigned to it. If given an input path that is a directory, MapReduce reads all files within the input directory and processes them.

- You can explore the Hadoop MapReduce API in the link: http://hadoop.apache.org/docs/r3.1.0/api/, and find "**org.apache.hadoop.mapreduce**"

Experiments 1: (15 points)

The WordCount program counts the number of occurrences of every distinct word in a document.

1. Download the sample code, WordCount.java, from Canvas and compile the program (further instruction can be found at Section 9 & 10 in Lab 1 instruction)
2. Download Shakespeare dataset from Cybox, and move it to HDFS
   https://iastate.box.com/s/pmzo1k6n1vewxl581rmoeapavzv0gzo1
3. Run jar file by
   hadoop jar <location_to_jar > <class_name> <HDFS_location_to_dataset> <Output_location>
   For instance,
   hadoop jar ~/Downloads/WordCount.jar WordCount /data/shakespeare /lab2/output

   Note that there is a space between each parameter.

Other information:

Make sure that the output location is empty before running a new job. Otherwise, the job will fail. There is one output file generated in the output location for every reducer. Screenshot the first 10 lines and mention them in the report.

Experiments 2: (50 points)

A bigram is a contiguous sequence of two words within the same sentence. For instance, the text “This is a car. This car is fast.” Has the following bigrams: “this is”, “is a”, “a car”, “this car”, “car is”, “is fast”. Note that the two words within a bigram must be a part of the same sentence. For example, “car this” is not a bigram since these words are not a part of the same sentence. Also, note that you need to convert all upper-case letters to lower-case first.

Task: Write a Hadoop program to identify the 10 most frequently occurring bigrams, along with their frequencies of occurrence.

You can assume that each call to the Map method receives one line of the file as its input. Note that it is possible for a bigram to span two lines of input but they are in two mapper input splits; you can ignore such cases and you do not have to count such bigrams. Remove any punctuation marks such as “,”, “.”, “?”, “!” etc from the input text. It can be done in the mapper, regex:

```
$  String.replaceAll("[^a-z0-9]", "")
```

Question: Think about how you might be able to get around the fact that bigrams might span lines of input. Briefly describe how you might deal with that situation? (5 points)

Hint: In this task, you might need multiple rounds of MapReduce and there is a sample code, TwoRoundWordCount.java, on Canvas.

Make sure that the output path of each MapReduce job is a unique directory that does not conflict with the output path of another job.

Also make sure that your temp directory (of first round MR) and the final output location (of second round MR) are empty before running a new job, otherwise job will fail.

Skeleton code (Driver.java) is provided to help you to start with the exercise. The code is well commented so as to help you understand it. Read the comments in detail and try to understand the code. Then you can build on this code to write your program.

You can use the Shakespeare corpus as the testing input to your program. Once your program runs successfully on this input, you can try to run your program on larger input files – Gutenberg dataset(https://iastate.box.com/s/mzzo9eisvqdutdkcd70xu49d1bhi8wqq)

Please submit the source code and the outputs for both datasets: Shakespeare and Gutenberg. Your output must include the top 10 most frequent bigrams and their frequencies of occurrence.