Tutorial:

Introduction to Hadoop MapReduce

Sayed Hadi Hashemi (last update: February 2018)

1. Overview

Welcome to the MapReduce tutorial. This tutorial covers the critical skills needed to develop a Hadoop MapReduce application.

Objectives

Upon completing this tutorial, students will be able to:

- perform basic operations on HDFS
- Develop a simple MapReduce applications
- Run MapReduce applications and examine the output
- Make modifications in MapReduce applications

Structure

This guide is designed as a set of step-by-step instructions for hands-on exercises. It teaches you how to operate the MapReduce service in a functional test environment through a series of examples. Exercises should be completed in the order described in the following steps:

1. Download some input datasets
2. Prepare a HDFS and upload dataset on it
3. Develop a simple "Word Count" application and build it
4. Run the application on Hadoop and examine the output
5. Modify the "Word Count" to discard common words
6. Rebuilt, and rerun the modified application
7. Do it yourself modification

2. Requirements

This tutorial is designed to work on the sample Docker image that we build following the "Tutorial: Docker installation" in Week 4. The virtual machine will have **Apache Hadoop 2.9.0**, **Apache Spark 2.2.1 (**for Apache Hadoop 2.7 and later**),** and **Apache Storm 1.0.5** installed.

3. Prepare the HDFS

MapReduce needs all the input and output files to be located on HDFS. In this section, we will first create a folder for this tutorial and download some text file as the input dataset. Then we create necessary folders on HDFS. Lastly, we copy the dataset to the HDFS. For the input dataset some masterpieces by William Shakespeare (Hamlet, Macbeth, Othello, and Romeo & Juliet) will be downloaded directly from the Gutenberg Project.

step 1: run the sample Docker image (please follow "Tutorial: Docker installation" in week 4)

```
$ docker run -it sample_image.v1 bin/bash

root@f9922c3fe307:/#
```

step 2: create a folder for this tutorial and change the current folder:

```
1  # mkdir mapreduce-tutorial
2  # cd mapreduce-tutorial
```

step 3: create a folder for the input datasets and download the following Shakespeare books:

```
1  # mkdir dataset
2
3  # wget -c http://www.gutenberg.lib.md.us/2/2/6/2264/2264.txt -P dataset
4  # wget -c http://www.gutenberg.lib.md.us/2/2/6/2265/2265.txt -P dataset
5  # wget -c http://www.gutenberg.lib.md.us/2/2/6/2266/2266.txt -P dataset
6  # wget -c http://www.gutenberg.lib.md.us/2/2/6/2267/2267.txt -P dataset
```

step 4: create input folder on HDFS and upload input datasets to HDFS:

```
1  # hadoop fs -mkdir -p /tutorial/input
2  # hadoop fs -put ./dataset/* /tutorial/input
```

4. Hello World (Word Count)

The Word Count application is the canonical example in MapReduce. It is a straightforward application of MapReduce, and MapReduce can handle word count extremely efficiently. In this particular example, we will be doing a word count within Shakespeare's books.

step 1: import the WordCount.java file from /etc/apt/WordCount.java

```
1  # mv /etc/apt/WordCount.java ./WordCount.java
```

step 2: create a temporary folder for compiling the code:

```
1  # mkdir build
```

step 3: compile the source code using following commands:

```
1  # hadoop com.sun.tools.javac.Main WordCount.java -d build
2  # jar -cvf WordCount.jar -C build/ ./
3
```

step 4: run the application using the following command:

```
# hadoop jar WordCount.jar WordCount /tutorial/input /tutorial/output
```

step 5: The output of the application will be on the HDFS. To see the output, you may use the following command:

```
# hadoop fs -cat /tutorial/output/part*
```

step 6 (optional; in case of runtime errors) If you encounter any compilation error (possibly due to typos), you need to first edit the source file to fix the problems, and then compile the code again using the direction above. Additionally, the output folder on HDFS should be cleaned before a new run using the following command:

```
# hadoop fs -rm -r -f /tutorial/output
```

5. Makes Modifications

Let's make some changes to the Word Count application. As you might have already noticed, words are not correctly tokenized. This happens because the default function for tokenization in Java only uses a limited number of splitter characters. Furthermore, there are a few common English words that are not interesting enough to be included in the output. In this section, we will fix these two problems.

step 1: open the source code WordCount.java using 'vim'

step 2: edit TokenizerMapper class in WordCount.java to match with following code:

```
public static class TokenizerMapper
        extends Mapper<Object, Text, Text, IntWritable>{

         List<String> commonWords = Arrays.asList("the", "a", "an", "and", "of",
           "to", "in", "am", "is", "are", "at", "not");

      private final static IntWritable one = new IntWritable(1);
      private Text word = new Text();

      public void map(Object key, Text value, Context context
                      ) throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line, " \t,;.?!-:@[](){}_
          */");
        while (tokenizer.hasMoreTokens()) {
          String nextToken = tokenizer.nextToken();
          word.set(nextToken);
          if (!commonWords.contains(nextToken.trim().toLowerCase())){
            context.write(word, one);
          }
        }
      }
    }
```

step 3: rebuild the code:

```
# rm –rf ./build/* ./WordCount.jar
# hadoop com.sun.tools.javac.Main WordCount.java -d build
# jar -cvf WordCount.jar -C build/ ./
```

step 4: clear the output folder, run the application, and check the results:

```
# hadoop fs -rm -r /tutorial/output
# hadoop jar WordCount.jar WordCount /tutorial/input /tutorial/output
# hadoop fs -cat /tutorial/output/part*
```

6. Do it yourself

One other problem with the current output is that it is too long. In this section, we will develop a new application which only shows the top 10 most commonly used words.

step 1: Clean up previous build file:

```
# rm -rf ./build/* ./*.jar
```

step 2: Import the TopWords.java file from /etc/apt/TopWords.java

```
# mv /etc/apt/TopWords.java ./TopWords.java
```

step 3: Try to compile and run this application yourself.

7. Questions to Think About

1. Why should data be on HDFS?
2. Why should the "output" folder be removed each time?
3. In the “Top Word” example, can you develop a new code which reuses (not copypaste) the “Word Count” code?