INFSCI 2750 - Mini Project 01

Yichi Zhang (yiz141@pitt.edu)
Quan Zhou (quz3@pitt.edu)

Part 1

With the documentation from the Apache Hadoop website and the in-class tutorial, we can set up the Hadoop cluster step-by-step.

The cluster on the DigitalOcean VMs are running properly. We can monitor the cluster's status and performance through the following webpages.

Monitor	Link
HDFS NameNode	http://159.89.43.89:9870
DataNode on master	http://159.89.43.89:9864
DataNode on slave	http://159.89.43.152:9864
ResourceManager of Yarn	http://159.89.43.89:8088
NodeManager on master	http://159.89.43.89:8042
NodeManager on slave	http://159.89.43.152:8042

From the Yarn ResourceManager page, we can see that the cluster has finished both the WordCount and the PI MapReduce program. And it also successfully finished the log analysis job in part 4.

One issue we encountered is with the split strategy of Yarn. Yarn will split the Map task into the number of blocks occupied by the input. If the number of files in the input exceeds a certain number, or the input file is so big that it's split into multiple parts, the ResourceManager tends to fail. In some cases, the NodeManager on master fails before the ResourceManager.

We observed the process of the failure from the ResourceManager and discovered that one of the main reasons of failure would be memory issues. Too many containers created by Yarn eventually take up all the memory and crashes the process. Modifying the Yarn configuration does help with the failure threshold but with larger input, the managers still fail easily with OOM yielding from the log.

One possible solution is to use the ResourceManager high availability functionality offered by the Yarn framework. Setting up multiple ResourceManagers can solve the problem for the most of it.

Part 2 : Docker

The required code for this part is provided in the `part_2_docker` folder. The file `Dockerfile` serves as the main build script for the image building process. The folder `config` contains several Hadoop configuration files as well as the `bootstrap.sh` file which starts the Hadoop services when the image starts up running. To create a Docker image with Hadoop built in, simply open any terminal and cd to the folder, run the command:

```
$ docker build -t hadoop .
```

After the build process is finished, run the following command to start the image in a container:

```
$ docker run --name hadoop
    -p 9870:9870 \
    -p 9868:9868 \
    -p 9864:9864 \
    -p 50060:50060 \
    -p 50030:50030 \
    -p 19888:19888 \
    -p 10033:10033 \
    -p 8032:8032 \
    -p 8030:8030 \
    -p 8088:8088 \
    -p 8033:8033 \
    -p 8042:8042 \
    -p 8188:8188 \
    -p 8047:8047 \
    -p 8788:8788 \
    -ti hadoop
```

For the convenience of testing, we implemented a simple test case and demo in the folder `demo`. When Docker finishes the start up process, we can start the demo by running:

```
$ sh ${HADOOP_HOME}/demo.sh
```

This shell script runs a WordCount job on the two text files `testfile1.txt` and `testfile2.txt` and prints the result in the terminal.

Part 3: N-Gram

The `NGram.java` file in `part_3_ngram` should first be compiled by running:

```
$ bin/hadoop com.sun.tools.javac.Main NGram.java
$ jar cf ngram.jar NGram*.class
```

When finished, we can run the MapReduce job by:

```
$ bin/hadoop jar ngram.jar NGram <input_file> <output_folder> <n>
```

For example:

```
$ bin/hadoop jar ngram.jar NGram input output 2
```

We've provided two test case outputs in `testcase_output` with bigram process. `helloworld` give the n-gram count for the simple string `helloworld`, while `access_log` shows the n-gram count of the log file from part 4. Note that the 500 megabyte log file took us about 8 minutes to finish the whole MapReduce job on a single node.

Part 4

The four JAVA files in `part_4_log_analysis` are the source code of the solutions to the four problems, respectively. All the files can be compiled and run to get the desired output in the same manner as in part 3.

Problem 1

```
$ bin/hadoop com.sun.tools.javac.Main LogAnalysis1.java
$ jar cf log1.jar LogAnalysis1*.class
$ bin/hadoop jar log1.jar LogAnalysis1 input output1
```

The output for problem 1, provided in `output1` is:

```
/assets/img/home-logo.png 98744
```

Note there are a few similar `GET` requests asking for the same resource. We can obtain those numbers by slightly changing the String matching criteria from `equals()` to `contains()`, which gives us the following output, provided in `output1_contains` is:

```
//assets/img/home-logo.png 1
/assets/img/home-logo.png 98744
/assets/img/home-logo.png? 1
/assets/img/home-logo.png?= 14
/assets/img/home-logo.png?id=606 1
/assets/img/home-logo.png?id=613 4
/assets/img/home-logo.png?id=620 1
/assets/img/home-logo.png?id=630 1
/assets/img/home-logo.png?id=637 4
/assets/img/home-logo.png?id=651 1
/assets/img/home-logo.png?id=654 1
/assets/img/home-logo.png?id=670 1
/assets/img/home-logo.png?id=680 2
```

Problem 2

```
$ bin/hadoop com.sun.tools.javac.Main LogAnalysis2.java
$ jar cf log2.jar LogAnalysis2*.class
$ bin/hadoop jar log2.jar LogAnalysis2 input output2
```

The output for problem 2, provided in `output2` is:

```
10.153.239.5 547
```

Problem 3

```
$ bin/hadoop com.sun.tools.javac.Main LogAnalysis3.java
$ jar cf log3.jar LogAnalysis3*.class
$ bin/hadoop jar log3.jar LogAnalysis3 input output3
```

Our implementation of this problem is slightly different from the previous ones. We implemented different classes for the Reducer and the Combiner. This is due to the fact that we have to find the largest value in Reducer whereas we are still doing the same in the Combiner.
The output for problem 3, provided in `output3` is:

```
/assets/css/combined.css 117348
```

Problem 4

```
$ bin/hadoop com.sun.tools.javac.Main LogAnalysis4.java
$ jar cf log4.jar LogAnalysis4*.class
$ bin/hadoop jar log4.jar LogAnalysis4 input output4
```

With the similar implementation in problem 3, the output, provided in `output4` is:

```
10.216.113.172 158614
```