MMD PROJECT - MINING OF MASSIVE DATASETS

Submitted by:
PRIYANK SONKIYA - 17DCS009

The assignment is divided in four parts –

1. CREATING THE ENVIRONMENT
2. PREPARING THE DATA SET
3. RUNNING A JOB IN HADOOP
4. UNDERSTANDING THE SOURCE CODE

CREATING THE ENVIRONMENT

Pre-requisites -

A. UBUNTU 18.04
B. HADOOP 2.8.5
C. ORACLE JDK 1.8

```
priyank@priyank-VirtualBox:~$ tar -xvf jdk-8u241-linux-x64.tar -C /opt
jdk1.8.0_241/
jdk1.8.0_241/jre/
jdk1.8.0_241/jre/plugin/
jdk1.8.0_241/jre/plugin/desktop/
jdk1.8.0_241/jre/plugin/desktop/sun_java.png
tar: jdk1.8.0_241/jre/plugin/desktop/sun_java.png: Cannot open: File exists
jdk1.8.0_241/jre/plugin/desktop/sun_java.desktop
tar: jdk1.8.0_241/jre/plugin/desktop/sun_java.desktop: Cannot open: File exists
jdk1.8.0_241/jre/Welcome.html
tar: jdk1.8.0_241/jre/plugin/desktop: Cannot utime: Operation not permitted
tar: jdk1.8.0_241/jre/plugin: Cannot utime: Operation not permitted
tar: jdk1.8.0_241/jre/Welcome.html: Cannot open: File exists
jdk1.8.0_241/jre/THIRDPARTYLICENSEREADME.txt
tar: jdk1.8.0_241/jre/THIRDPARTYLICENSEREADME.txt: Cannot open: File exists
```

a. The above command unpacks the jdk- package (java archive) in /opt folder.

```
priyank@priyank-VirtualBox:~$ update-alternatives --install /usr/bin/java java /
opt/jdk1.8.0_241/bin/java 100
update-alternatives: error: unable to create file '/var/lib/dpkg/alternatives/ja
va.dpkg-tmp': Permission denied
priyank@priyank-VirtualBox:~$ sudo update-alternatives --install /usr/bin/java j
ava /opt/jdk1.8.0_241/bin/java 100
[sudo] password for priyank:
priyank@priyank-VirtualBox:~$ sudo update-alternatives --install /usr/bin/javac
javac /opt/jdk1.8.0_241/bin/javac 100
```

b. To set the jdk 1.8 update 241 as the default jvm we use above commands.

c. Now we download the Hadoop 2.8.5 from the official website and extract/unpack it.

```
priyank@priyank-VirtualBox:~$ tar -xzvf hadoop-2.8.5.tar.gz
```

d. Now we edit the bashrc file and source it, also we edit the Hadoop-env.sh file and set the environment variable. Received from step b.

```
hadoop@priyank-VirtualBox:~$ cat hadoop-2.8.5/etc/hadoop/hadoop-env.sh
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements.  See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership.  The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License.  You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# Set Hadoop-specific environment variables here.

# The only required environment variable is JAVA_HOME.  All others are
# optional.  When running a distributed configuration it is best to
# set JAVA_HOME in this file, so that it is correctly defined on
# remote nodes.

# The java implementation to use.
export JAVA_HOME=/opt/jdk1.8.0_241
```

e. Further we make a few changes in the core-site.xml, mapred-site.xml and yarn-site.xml as explained in the website which has been given credit to in the end. By default, Hadoop is configured to run in a non-distributed mode, as a single Java process.

f. Once all of this is done then we start-dfs.sh, start-yarn.sh. These commands start our hdfs (storage) and mapred cluster.

```
hadoop@priyank-VirtualBox:~$ start-dfs.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /home/hadoop/hadoop-2.8.5/logs/hadoop-h
adoop-namenode-priyank-VirtualBox.out
localhost: starting datanode, logging to /home/hadoop/hadoop-2.8.5/logs/hadoop-h
adoop-datanode-priyank-VirtualBox.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /home/hadoop/hadoop-2.8.5/logs/h
adoop-hadoop-secondarynamenode-priyank-VirtualBox.out
hadoop@priyank-VirtualBox:~$ start-yarn.sh
starting yarn daemons
starting resourcemanager, logging to /home/hadoop/hadoop-2.8.5/logs/yarn-hadoop-
resourcemanager-priyank-VirtualBox.out
localhost: starting nodemanager, logging to /home/hadoop/hadoop-2.8.5/logs/yarn-
hadoop-nodemanager-priyank-VirtualBox.out
```

g. To check if the cluster is running properly we check by -

```
hadoop@priyank-VirtualBox:~$ /opt/jdk1.8.0_241/bin/jps
2753 NameNode
3091 SecondaryNameNode
3365 NodeManager
2885 DataNode
3691 Jps
3246 ResourceManager
```

This means that all our resources are running properly.
Our cluster is ready for storage and computation.

PREPARING THE DATASET

A. Download The Project Gutenberg EBook of Debian GNU/Linux: Guide to Installation and Usage by John Goerzen and Ossama Othman from http://www.gutenberg.org/ebooks/ in .txt format.

B. Appended the necessary strings into the book as asked.

```
hadoop@priyank-VirtualBox:~$ echo "CSE3152 17DCS009" >> book.txt
hadoop@priyank-VirtualBox:~$ echo "CSE3152 17UCS118" >> book.txt
```

RUNNING THE JOB IN HADOOP

We have the program WordCount.java available on - https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Example:_WordCount_v1.0
That is on the official Apache Hadoop website.

- Compile this program using

```
/bin/hadoop com.sun.tools.javac.Main WordCount.java
```

- Now we will create a jar out of this

```
jar cf wc.jar WordCount*.class
```

  The WordCount.* signifies that all the files that are the source code and the compiled files all will be used in creating a jar.

- Now we have the appended book book.txt, we make an input and output directories for our program -

```
bin/hdfs dfs -mkdir input
bin/hdfs dfs -mkdir output
```

- Now we will put our desired file books.txt in our input directory,
  By command

```
/home/priyank/hadoop-2.8.5/input/  book.txt
```

- Running the final command -

```
bin/hadoop jar wc.jar
WordCount/home/priyank/hadoop-2.8.5/input/home/pri
yank/hadoop-2.8.5/output
```

  This command will display all the words and their counts from the file book.txt which is kept in the distributed file system of the Hadoop cluster.

The output directory contains two files part-r-00000 and _SUCCESS, out of which the file part-r-00000 file contains all the word count as from the input text file -

```
hadoop@priyank-VirtualBox:~/hadoop-2.8.5/output$ cat part-r-00000.txt | grep "17
UCS"
17UCS118        1
hadoop@priyank-VirtualBox:~/hadoop-2.8.5/output$ cat part-r-00000.txt | grep "17
DCS"
17DCS009        1
hadoop@priyank-VirtualBox:~/hadoop-2.8.5/output$
```

The occurrences of our roll numbers in the output file.
Also, CSE3152 should occur 2 times - hence,

```
hadoop@priyank-VirtualBox:~/hadoop-2.8.5/output$ cat part-r-00000.txt | grep "CS
E3152"
CSE3152 2
hadoop@priyank-VirtualBox:~/hadoop-2.8.5/output$
```

Also a picture of how the output file looks like -

```
hadoop@priyank-VirtualBox:~/hadoop-2.8.5/output$ cat part-r-00000.txt
!       1
"       1
"AS     1
"AS-IS".        1
"Defects".      1
"PROJECT        3
"Program",      1
"Project        2
"Project").     1
"Right  1
"Small  5
"any    1
"change"        1
"copyright      1
"copyright"     1
"hello" 2
"legal  1
"modification".)        1
"public 1
"small  1
"work   1
"you".  1
#       6
#6527]  1
#<dump  1
#debian 1
$       30
$.      1
$15     1
$2      1
```

The file shows the occurrence of each word and the count of the frequency.
For detailed view the output file link- https://docs.google.com/document/d/17QYwbU2ucN1XNrDPpgdFp68KYqc9vj9dwcopcwIcf9Q/edit?usp=sharing contains the output file.

UNDERSTANDING THE SOURCE CODE

The MapReduce framework operates exclusively on <key, value> pairs, that is, the framework views the input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job, conceivably of different types.

(input) <k1, v1> -> **map** -> <k2, v2> -> **combine** -> <k2, v2> -> **reduce** -> <k3, v3> (output)

The WordCount application is quite straight-forward.

```
Job job = Job.getInstance(conf, "word count");
job.setJarByClass(WordCount.class);
```

A Job object is created and is assigned a name word count. The method job.setJarByClass(WordCount.class), identifies for each node the jar file containing Mapper and Reducer classes.

The Mapper implementation, via the map method, processes one line at a time, as provided by the specified TextInputFormat. It then splits the line into tokens separated by whitespaces, via the StringTokenizer, and emits a key-value pair of < <word>, 1>.

```
public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedException {
  StringTokenizer itr = new StringTokenizer(value.toString());
  while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
  }
}
```

The Hadoop MapReduce framework spawns one map task for each InputSplit generated by the InputFormat for the job.

```
job.setMapperClass(TokenizerMapper.class);
```

Mapper implementations are passed to the job via Job.setMapperClass(Class) method. The framework then calls

map(WritableComparable, Writable, Context) for each key/value pair in the InputSplit for that task.

```
job.setCombinerClass(IntSumReducer.class);
```

WordCount also specifies a combiner. Hence, the output of each map is passed through the local combiner (which is same as the Reducer as per the job configuration) for local aggregation, after being sorted on the *keys*. Users can optionally specify a combiner, via job.setCombinerClass(Class).

The Reducer implementation, via the reduce method, just sums up the values, which are the occurrence counts for each key (i.e. words in this example).

```
public void reduce(Text key, Iterable<IntWritable> values,
                   Context context
                   ) throws IOException, InterruptedException {
  int sum = 0;
  for (IntWritable val : values) {
    sum += val.get();
  }
  result.set(sum);
  context.write(key, result);
}
```

```
job.setReducerClass(IntSumReducer.class);
```

Reducer implementations are passed the Job for the job via the Job.setReducerClass(Class) method and can override it to initialize themselves. The framework then calls reduce(WritableComparable, Iterable<Writable>, Context) method for each <key, (list of values)> pair in the grouped inputs.

The number of reduces for the job is set by the user via Job.setNumReduceTasks(int).

```
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

FileInputFormat indicates the set of input files and where the output files should be written

References:

- https://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/examples/WordCount.TokenizerMapper.html
- https://hadoop.apache.org/docs/r2.4.1/api/org/apache/hadoop/mapreduce/Mapper.html
- https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Reducer.html
- https://stackoverflow.com/questions/29063844/key-of-object-type-in-the-hadoop-mapper
- https://www.javatpoint.com/string-tokenizer-in-java
- https://hadoop.apache.org/docs/r2.4.1/api/org/apache/hadoop/io/Text.html
- https://www.quora.com/What-is-the-use-of-Context-object-in-Hadoop
- https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Example:_WordCount_v1.0