ID2221: Data Intensive Computing
Lab 1 - Apache Hadoop

(Updated 2017-08-30)

In this lab you will practice the basics of the data intensive programming by setting up HDFS and Hadoop MapReduce and then implementing a simple applications. We will be using Docker containers in most of the labs so you will also learn the basics of how to use Docker.

Part 1 - Setting up the Work Environment

Step 1 - Install Docker

Linux

The following instructions is tested on **Ubuntu 17.04**

```
sudo apt update && sudo apt upgrade
sudo apt install docker.io
```

To be able to run docker commands without “sudo” you must **add** your linux user to the **docker group**

```
sudo usermod -aG docker $USER
```

Then you need to **logout** and login for the change to take effect

Window and Mac

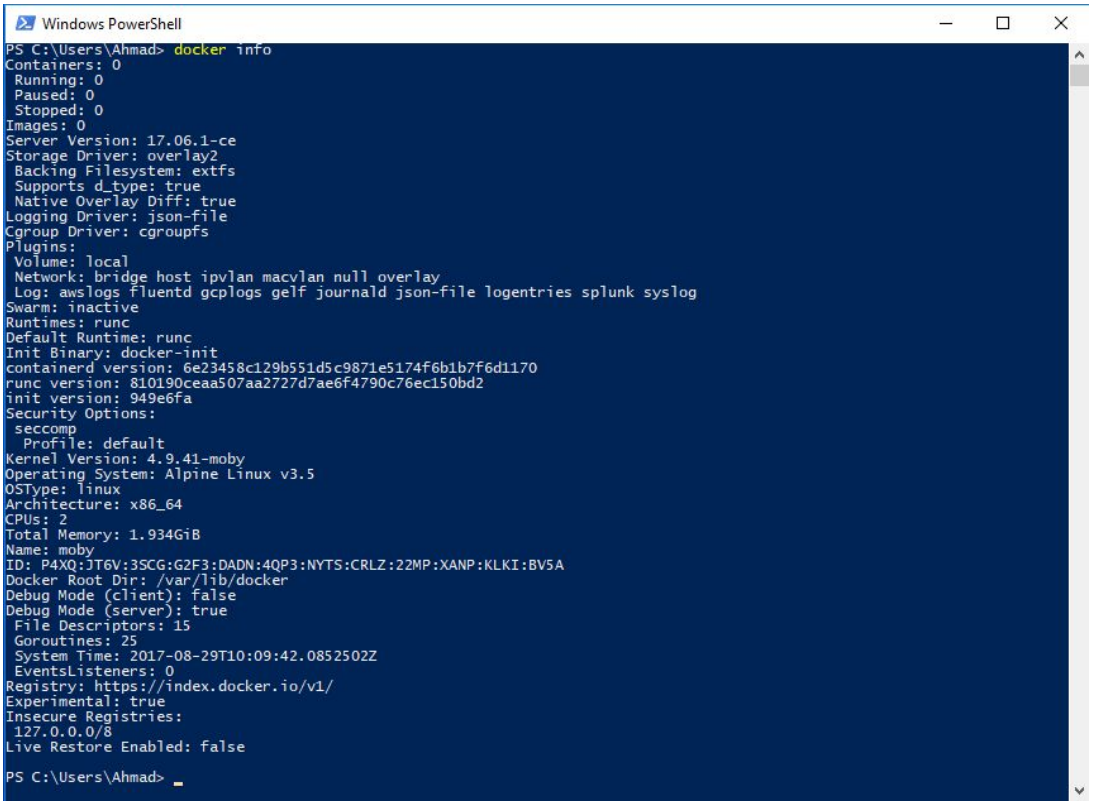
Download in install Docker

- Windows:
  - Docker (stable) for Windows: https://download.docker.com/win/stable/Docker%20for%20Windows%20Installer.exe
  - More info: https://docs.docker.com/docker-for-windows/install/#download-docker-for-windows
- Mac:
  - Docker (stable) for Mac: https://download.docker.com/mac/stable/Docker.dmg
  - More info: https://docs.docker.com/docker-for-mac/install/#download-docker-for-mac
- Alternatively you can download the docker-toolbox that contains extra tools (not used in labs) https://www.docker.com/products/docker-toolbox

Step 2 - Verify Your Installation

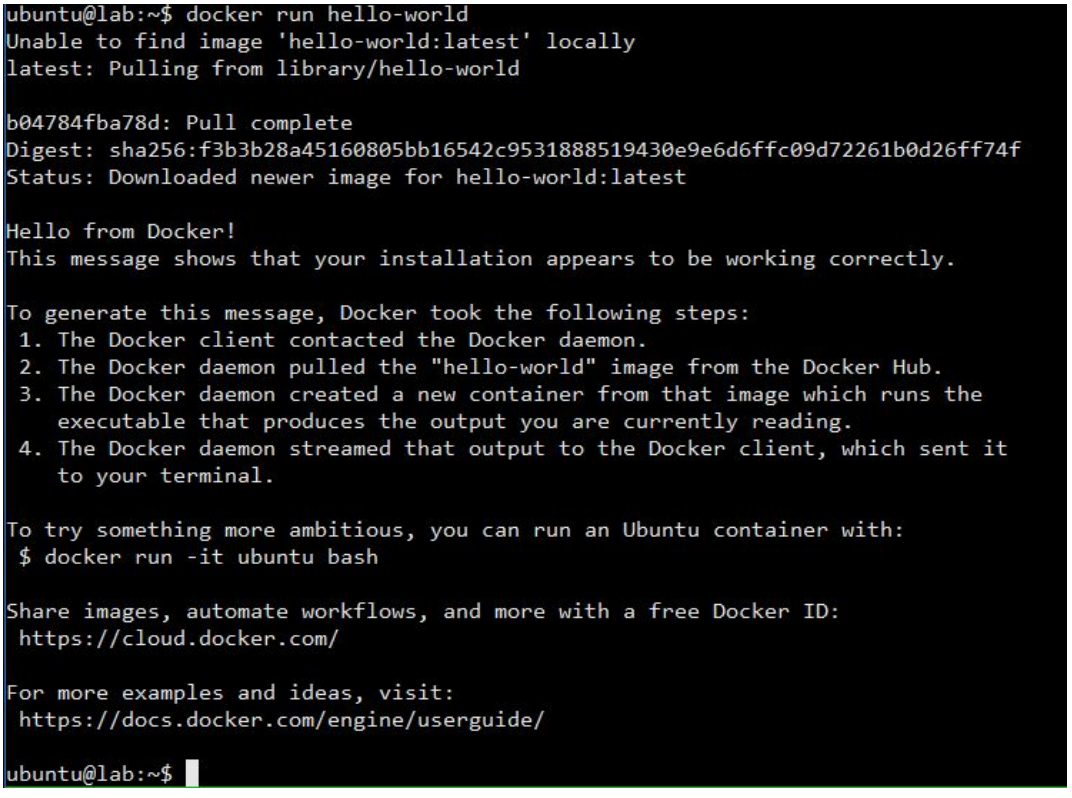
Check that docker is running by displaying basic docker information:

```
docker info
```



Try to run a “Hello World” test image

```
docker run hello-world
```



Step 3 - Start the Hadoop Docker Image

There is no official Apache Hadoop image so we will be using a **simple** Hadoop image from https://github.com/sequenceiq/hadoop-docker

> Alternatively you can try the **Cloudera** Hadoop Distribution which contains many other tools https://www.cloudera.com/documentation/enterprise/5-6-x/topics/quickstart_docker_container.html#cloudera_docker_container

Pull the Hadoop image. Note the image version (2.7.1). You can instead pull latest, but maybe it is better to have the same versions for the lab.

This will take some time depending on your connection speed.

```
docker pull sequenceiq/hadoop-docker:2.7.1
```

It is good practice to keep the container **stateless** by keeping your files outside the container and by not installing additional packages. This way you can easily get newer versions of the container and run your code on other machines.

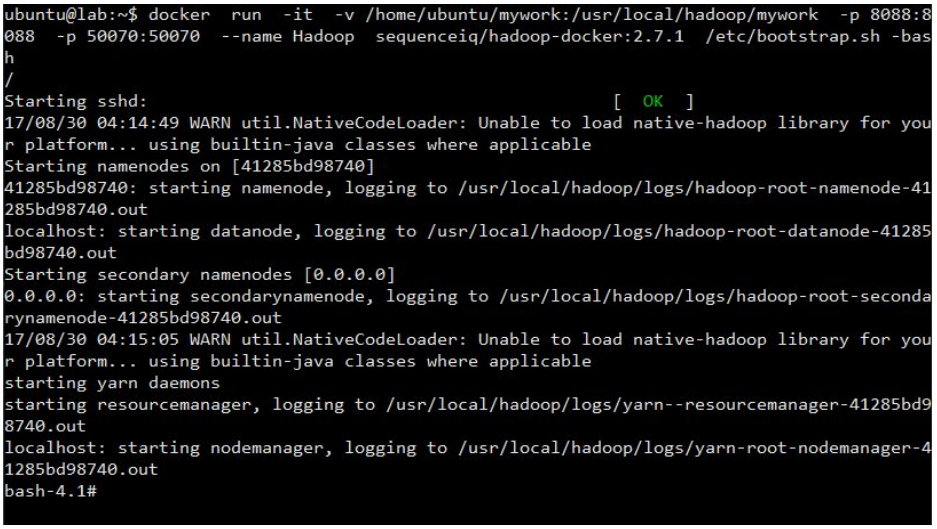
One way to achieve this is by creating a folder that will contain your files. We will mount this folder later inside the container.

```
mkdir mywork
```

Now we run the Hadoop container using the image we just pulled.

```
docker run -it -v /home/ubuntu/mywork:/usr/local/hadoop/mywork -p 8088:8088 -p 50070:50070 --name Hadoop sequenceiq/hadoop-docker:2.7.1 /etc/bootstrap.sh -bash
```

This should launch the Hadoop container and present you with the container's command prompt.



The arguments for the docker command above are as follows

- **run:** this tells docker to run a *command* in a *new container.* The command we are running is "/etc/bootstrap.sh -bash". This command is in the Hadoop docker container image we just pulled. It bootstraps the Hadoop and HDFS services and starts "bash" as in the screenshot above. The run command also needs the name of the new container image, in our case this is "`sequenceiq/hadoop-docker:2.7.1`"
- **-it:** For interactive processes (like a shell), we must use -i -t together in order to allocate a *tty* (i.e., terminal) for the container process. -i -t is often written as -it.
    - -t: Allocate a pseudo-tty (terminal)
    - -i: Keep STDIN open even if not attached (We'll explain attach/detach later)
- **--name:** This is to give a meaningful name to our new running container. Otherwise it will be assigned a random name.
- **-p hostPort:containerPort:** If we want to access services running in the container, we must open corresponding ports. This is done by mapping the service port in the container "containerPort" to a free port in the host (host is your computer) hostPort. In our case, Hadoop WebUI is running on port 8088 (containerPort) and HDFS WebUI is running on port 50070 (containerPort). For simplicity we map these ports to the same port on the host but you can choose any other free ports on the host. To access the service, you open a web browser and point it to localhost:<hostPort>. For example http://localhost:50070/ to access HDFS WebUI.
- **-v:** Mount a volume. This allows us to access our files from the container. Here we mount a folder that will contain our files (on the host) to a folder in the container. The host folder in the example above is '/home/ubuntu/mywork'. **Please replace this path** to the path matching your folder. The "/usr/local/hadoop/mywork" path is where we will find our files in the container.

Step 4 - Verify Hadoop and HDFS

Now your Hadoop container should be running and you should see the command prompt.

Now we will run a simple MapReduce job that is included and check the output.

Check that all services are running using jps (jps - Java Virtual Machine Process Status Tool)

```
jps
```

```
715 NodeManager
260 DataNode
613 ResourceManager
1058 Jps
438 SecondaryNameNode
134 NameNode
```

Change to the hadoop folder

```
cd $HADOOP_PREFIX
```

Run the included MapReduce job

```
bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.1.jar grep input output 'dfs[a-z.]+'
```

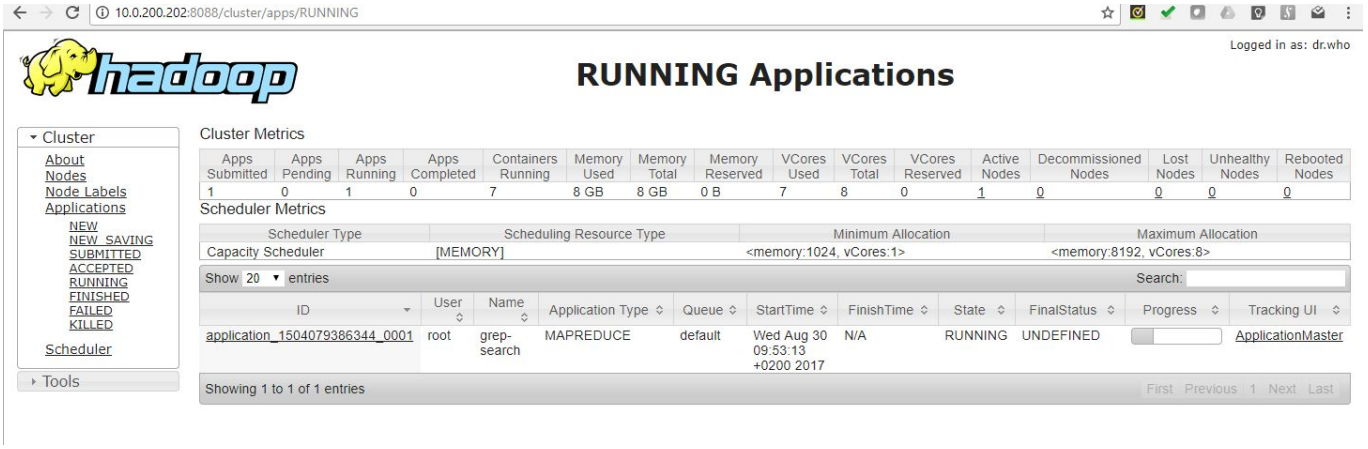
Display the output stored in HDFS

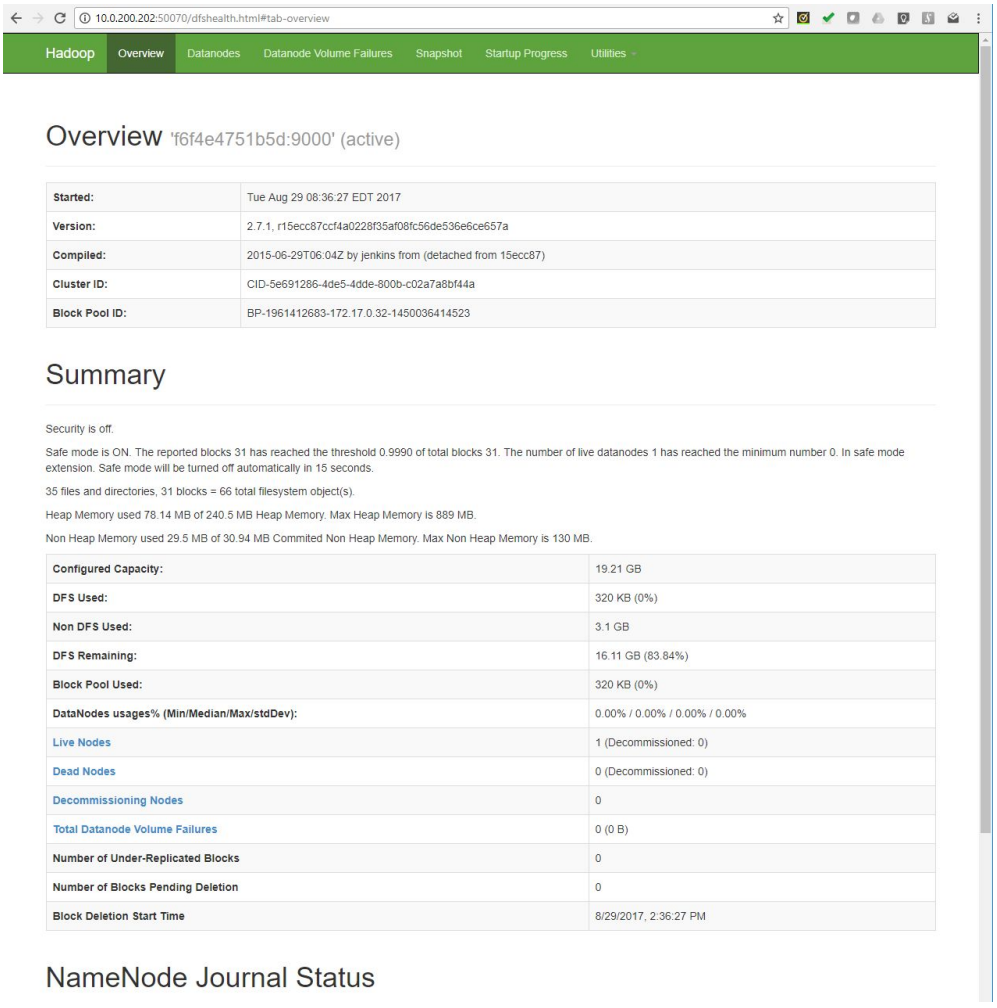
```
bin/hdfs dfs -cat output/*
```

Your output should be similar to the screenshot below

```
bash-4.1# bin/hdfs dfs -cat output/*
17/08/30 03:56:26 WARN util.NativeCodeLoader: Unable to load native-hadoop library for
 your platform... using builtin-java classes where applicable
6       dfs.audit.logger
4       dfs.class
3       dfs.server.namenode.
2       dfs.period
2       dfs.audit.log.maxfilesize
2       dfs.audit.log.maxbackupindex
1       dfsmetrics.log
1       dfsadmin
1       dfs.servers
1       dfs.replication
1       dfs.file
bash-4.1#
```

You can also track your running MapReduce job with the Hadoop WebUI at localhost:8088 and check your HDFS WebUI at localhost:50070





Step 5 - Managing Containers

Your Hadoop container should be running. Next we will go through some useful docker commands to manage your containers.

Detach from a Container

If you finished working with your container and want to return to the host prompt you can detach from the container by pressing <ctrl-p> then <ctrl-q>

Detach from container:

```
<ctrl-p> + <ctrl-q>
```

Detach means that the **container is still running.**

List Containers

Now we detached from the Hadoop container and back to the host. We can list running containers

```
docker ps
```

```
CONTAINER ID        IMAGE                           COMMAND                  CREATED              STATUS              PORTS
NAMES
d153797c768b        sequenceiq/hadoop-docker:2.7.1   "/etc/bootstrap.sh -b"   About a minute ago   Up 39 seconds       2122/tcp, 8030-8033/tcp, 8040/tcp,
8042/tcp, 19888/tcp, 49707/tcp, 50010/tcp, 0.0.0.0:8088->8088/tcp, 50020/tcp, 50075/tcp, 50090/tcp, 0.0.0.0:50070->50070/tcp   Hadoop
```

Or list all containers (exited, …)

```
docker ps -a
```

```
CONTAINER ID        IMAGE                           COMMAND                  CREATED             STATUS                      PORTS
NAMES
d153797c768b        sequenceiq/hadoop-docker:2.7.1   "/etc/bootstrap.sh -b"   6 minutes ago       Up 6 minutes                2122/tcp, 8030-8033/tcp,
8040/tcp, 8042/tcp, 19888/tcp, 49707/tcp, 50010/tcp, 0.0.0.0:8088->8088/tcp, 50020/tcp, 50075/tcp, 50090/tcp, 0.0.0.0:50070->50070/tcp   Hadoop

0716ca098c6e        hello-world                     "/hello"                 50 minutes ago      Exited (0) 50 minutes ago
reverent_turing
```

Delete Containers

We see the hello-world container we used to test our docker installation. Notice that it got a random name (in this example “reverent_turing”). We can delete this container as it is no longer needed.

```
docker rm reverent_turing
```

Attach to a Container

To attach back to our hadoop container we use docker attach. Remember that we named our container “--name Hadoop”

```
docker attach Hadoop
```

Stop a Container

If you type “exit” at the containers prompt, or run the command “docker stop Hadoop”, the container will stop running and exit.

```
docker stop Hadoop
docker ps -a
```

```
CONTAINER ID        IMAGE                              COMMAND                  CREATED             STATUS                         PORTS               NAMES
d153797c768b        sequenceiq/hadoop-docker:2.7.1   "/etc/bootstrap.sh -b"   22 minutes ago      Exited (137) 4 seconds ago                         Hadoop
```

Restarting a Stopped Container

Restart and then attach to the container

```
docker start Hadoop
docker attach Hadoop
```

Part 2 - Basic Examples

2.a - HDFS

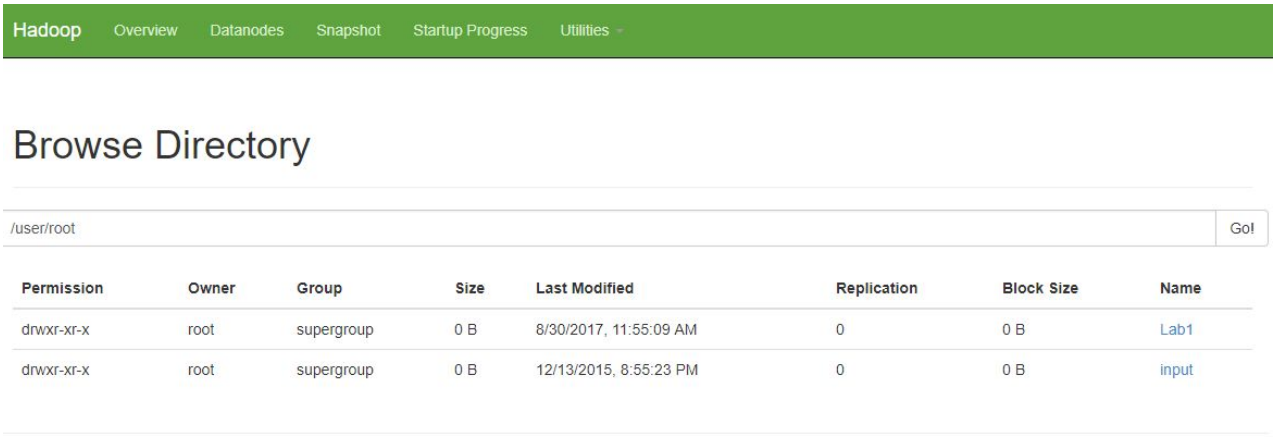
Run the Hadoop container if it is not running and then change to the Hadoop folder.

```
cd $HADOOP_PREFIX
```

Create an HDFS directory and call it “Lab1” in your home folder. Note that in this Hadoop image you are the root user and your home in HDFS is /user/root

```
bin/hdfs dfs -mkdir /user/root/Lab1
```

You can check your progress in the HDFS WebUI



Copy any text file in your work directory and rename it to BigText.

```
cp README.txt mywork/BigText
```

Upload the file to Lab1 folder we created in HDFS

```
bin/hdfs dfs -put mywork/BigText /user/root/Lab1
```

List the contents of the Lab1 directory

```
bin/hdfs dfs -ls /user/root/Lab1
```

Determine the size of the folder. Notice that the commands (ls, du, …) are very similar to the linux commands

```
bin/hdfs dfs -du -h /user/root/Lab1
```

Display the first 5 lines. Note that this is efficient even if you have very large file because “cat” will close the stream as soon as “head” finishes reading all the lines

```
bin/hdfs dfs -cat /user/root/Lab1/BigText | head -n 5
```

Make a copy of your file on HDFS

```
bin/hdfs dfs -cp /user/root/Lab1/BigText /user/root/Lab1/BigText_hdfscopy
```

Get a copy of your file on your local file system

```
bin/hdfs dfs -get /user/root/Lab1/BigText mywork/BigText_localcopy
```

Check the entire HDFS filesystem for errors

```
bin/hdfs fsck /
```

Delete a file from HDFS

```
bin/hdfs dfs -rm /user/root/Lab1/BigText_hdfscopy
```

Delete a folder (and recursively everything inside) in HDFS

```
bin/hdfs dfs -rm -r /user/root/Lab1
```

2.a MapReduce

Simple Word Count

WordCount is a simple application that counts the number of occurrences of each word in a given input set. Below we will take a look at the mapper and reducer in detail, and then we present the complete code and show how to compile and run it.

Word Count Mapper

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

   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 {
       StringTokenizer itr = new StringTokenizer(value.toString());
       while (itr.hasMoreTokens()) {
           word.set(itr.nextToken());
           context.write(word, one);
       }
   }
}
```

The `Mapper<Object, Text, Text, IntWritable>` refers to the data type of input and output key-value pairs specific to the mapper class or rather the map method. The signature of the Mapper class is `Mapper<Input Key Type, Input Value Type, Output Key Type, Output Value Type>`. In our example, the input to a mapper is a single line, so this Text forms the input value. The input key is a long value assigned in default based on the position of Text in input file. Our output from the mapper is

of the format (Word, 1) hence the datatype of our output key value pair is `<Text(String), IntWritable(int)>`.

In the `map` method, the first and second parameters refer to the data type of the input key/value pair to the map method. The third parameter is the output collector that does the job of taking the output data. With the output collector we need to specify the data types of the output key and value from the mapper. The fourth parameter is used to report the task status internally in Hadoop environment to avoid time outs.

`The` functionality of the map method is as follows:

1. Create an `IntWritable` variable `one` with value as 1.
2. Convert the input line in `Text` type to a `String`.
3. Use a tokenizer to split the line into words.
4. Iterate through each word and form key-value pairs as `(word, one)` and push it to the output collector.

Word Count Reducer

```
public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
   private IntWritable result = new IntWritable();

   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);
   }
}
```

In `Reducer<Text, IntWritable, Text, IntWritable>`, the first two parameters refer to data type of input key and value to the reducer and the last two refer to data type of output key and value. Our mapper emits output as, e.g., (apple, 1), (grapes, 1), (apple, 1), etc. This is the input for reducer so here the data types of key and value in java would be `String` and `int`, the equivalent in Hadoop would be `Text` and `IntWritable`. Also we get the output as (word, num. of occurrences) so the data type of output Key Value would be `<Text, IntWritable>`.

The input to `reduce` method from the mapper after the sort and shuffle phase would be the key with the list of associated values with it. For example here we have multiple values for a single key from our mapper like (apple, 1), (apple, 1), (apple, 1). These key-values would be fed into the reducer as (apple, [1, 1, 1]), which is a key and list of values (`Text` key, `Iterator<IntWritable>` values). The next parameter to the `reduce` method denotes the output collector of the reducer with the data type of output key and value.

The functionality of the reduce method is as follows:

1. Initialize a variable `sum` as 0.
2. Iterate through all the values with respect to a key and sum up all of them.
3. Push to the output collector, the key and the obtained sum as value.

Driver Class

In addition to mapper and reducer classes, we need a "driver" class to trigger the MapReduce job in Hadoop. In the driver class we provide the name of the job, output key-value data types and the mapper and reducer classes. Below you see the complete code of the "word count":

```
package lab1;

import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

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

        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 {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
   }

   public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
        private IntWritable result = new IntWritable();

        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);
        }
   }
```

```
   public static void main(String[] args) throws Exception {
       Configuration conf = new Configuration();
       Job job = Job.getInstance(conf, "word count");
       job.setJarByClass(WordCount.class);

       job.setMapperClass(TokenizerMapper.class);
       job.setCombinerClass(IntSumReducer.class);
       job.setReducerClass(IntSumReducer.class);

       job.setOutputKeyClass(Text.class);
       job.setOutputValueClass(IntWritable.class);

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

       System.exit(job.waitForCompletion(true) ? 0 : 1);
   }
}
```

Compile and Run The Code

Copy the “lab1” folder into your “mywork” folder on your computer (this will make files available in the Hadoop container). Alternatively, you can copy/paste the code above in a new file and create two input text files, “file0” and “file1”.

Start your Hadoop docker container as described in Part 1 if it is not running.

Upload the input files to HDFS

```
cd $HADOOP_PREFIX
bin/hdfs dfs -mkdir -p /user/root/lab1/wordcount/input
bin/hdfs dfs -put mywork/lab1/wordcount/input/file0 /user/root/lab1/wordcount/input
bin/hdfs dfs -put mywork/lab1/wordcount/input/file1 /user/root/lab1/wordcount/input
bin/hdfs dfs -ls /user/root/lab1/wordcount/input
```

Compile the code and create a jar file

```
cd mywork/lab1/wordcount

mkdir classes

javac -classpath
$HADOOP_PREFIX/share/hadoop/common/hadoop-common-2.7.1.jar:$HADOOP_PREFIX/share/hado
op/mapreduce/hadoop-mapreduce-client-core-2.7.1.jar:$HADOOP_PREFIX/share/hadoop/comm
on/lib/commons-cli-1.2.jar -d classes src/lab1code/WordCount.java

jar -cvf wordcount.jar -C classes/ .
```

Run the application

```
cd $HADOOP_PREFIX
bin/hadoop jar mywork/lab1/wordcount/wordcount.jar lab1code.WordCount
/user/root/lab1/wordcount/input /user/root/lab1/wordcount/output
```

Check the output in HDFS

```
bin/hdfs dfs -ls /user/root/lab1/wordcount/output

bin/hdfs dfs -cat /user/root/lab1/wordcount/output/part-r-00000
Bye     1
Goodbye 1
Hadoop  2
Hello   2
World   2
```

Part 3 - Exercise: Find Top Ten Users

Given the list of user information, print out the top ten users based on their reputation. In your code, each mapper determines the top ten records of its input split and outputs them to the reduce phase. The mappers are filtering their input split to the top ten records, and the reducer is responsible for the final ten. Use `setNumReduceTasks` to configure your job to use only one reducer, because multiple reducers would shard the data and would result in multiple top ten sub-lists.

The input data of this part is in the `lab1/topten/input` folder.

The Mapper Code

You can use the Java `TreeMap` structure in the mapper to store the processed input records. A `TreeMap` is a subclass of `Map` that sorts is sorted based on the key. When processing a user record, we can use the reputation as the key. After all the records have been processed, the top ten records in the `TreeMap` are output to the reducers in the `cleanup` method. This method gets called once after all key-value pairs have been through map.

```
public static class TopTenMapper extends Mapper<Object, Text, NullWritable, Text> {
   // Stores a map of user reputation to the record
   private TreeMap<Integer, Text> repToRecordMap = new TreeMap<Integer, Text>();

   public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
       Map<String, String> parsed = <...>
       String userId = <...>
       String reputation = <...>

       // check that this row contains user data
       <...>

       //Add this record to our map with the reputation as the key
       repToRecordMap.<...>

       // If we have more than ten records, remove the one with the lowest reputation.
       if (repToRecordMap.size() > 10) {
           <...>
       }
   }

   protected void cleanup(Context context) throws IOException, InterruptedException {
       // Output our ten records to the reducers with a null key
       for (Text t : repToRecordMap.values()) {
           context.write(NullWritable.get(), t);
       }
   }
}
```

The Reducer Code

The reducer determines its top ten records in a way that's very similar to the mapper. Because we configured our job to have one reducer using `job.setNumReduceTasks(1)` and we used `NullWritable` as our key, there will be one input group for this reducer that contains all the potential top ten records.

The reducer iterates through all these records and stores them in a `TreeMap`. After all the values have been iterated over, the values contained in the `TreeMap` are flushed to the file system in descending order.

```
public static class TopTenReducer extends Reducer<NullWritable, Text, NullWritable, Text> {
   // Stores a map of user reputation to the record
   // Overloads the comparator to order the reputations in descending order
   private TreeMap<Integer, Text> repToRecordMap = new TreeMap<Integer, Text>();

    public void reduce(NullWritable key, Iterable<Text> values, Context context) throws IOException,
InterruptedException {
       for (Text value : values) {
           Map<String, String> parsed = <...>
           repToRecordMap.<...>

           // If we have more than ten records, remove the one with the lowest reputation
           if (repToRecordMap.size() > 10) {
               <...>
           }
       }

       // Sort in descending order
       for (Text t : repToRecordMap.descendingMap().values()) {
           // Output our ten records to the file system with a null key
           context.write(NullWritable.get(), t);
       }
   }
}
```

The Input Data

The input file, users.xml, is in XML format with the following syntax:

```
<row Id="-1" Reputation="1"
CreationDate="2014-05-13T21:29:22.820" DisplayName="Community"
LastAccessDate="2014-05-13T21:29:22.820"
WebsiteUrl="http://meta.stackexchange.com/"
Location="on the server farm" AboutMe="..;"
Views="0" UpVotes="506"
DownVotes="37" AccountId="-1" />
```

Parsing the XML File

The Mapper will process the xml file line by line. Each line contains a single user record in the format described above. The Map method will get this as a single String value. You will need to parse this string to extract the user's reputation. You can use the simple function below to do that.

```
public static Map<String, String> transformXmlToMap(String xml) {
   Map<String, String> map = new HashMap<String, String>();
   try {
       String[] tokens = xml.trim().substring(5, xml.trim().length() - 3)
               .split("\"");

       for (int i = 0; i < tokens.length - 1; i += 2) {
           String key = tokens[i].trim();
           String val = tokens[i + 1];

           map.put(key.substring(0, key.length() - 1), val);
       }
   } catch (StringIndexOutOfBoundsException e) {
       System.err.println(xml);
   }

   return map;
}
```

Be careful when processing the XML file in the Map method! Skip over the rows that doesn't contain user data (e.g., when Id == null)

```
<?xml version="1.0" encoding="utf-8"?>
<users>
    <row Id=...
    <row Id=...
    ...
    ...
</users>
```

The Output

The output should be the sorted top ten user rows from the input xml file. This should be similar to the following:

```
  <row Id="2452" Reputation="4503" CreationDate="2014-07-11T08:54:54.627" DisplayName="Aleksandr Blekh" LastAcce...
  <row Id="381" Reputation="3638" CreationDate="2014-06-08T06:51:59.140" DisplayName="Emre" LastAccessDate="2016...
  <row Id="11097" Reputation="2824" CreationDate="2015-08-05T12:46:59.553" DisplayName="Dawny33" LastAccessDate=...
...
...
```

Part 4 - Extra: Find Top Ten Words (Optional)

Problem:

Given a number of large text files, display the top ten words! (or top ten words larger than 2 characters)

Spoiler Alert! Think a bit first before reading the hint!

Hint:

One simple way to achieve this is by creating two MapReduce jobs. The first one is similar to the word count example, it will read the input files and will write the output as the count of each word to HDFS. Note that this count is similar to the user's reputation! The second MapReduce job will use the output of the first job as input, it will be similar to the top ten users example that will find the top words with highest counts and write the final output to HDFS.