Lab1 - MapReduce, HDFS, and HBase

1   Introduction

In this lab assignment, you will practice the basics of the data intensive programming by setting up HDFS, HBase and Hadoop MapReduce and implementing a simple application on them. Note that the following instruction is tested on a Linux operating system, so if you do not have Linux, you need to install it either on your machine or on a VirtualBox. You can download VirtualBox from its page. You can also find different ready to use Linux distribution images for VirtualBox here.

Section 2 and Section 3 of this document demonstrate how to install MapReduce, HDFS, and HBase, and work with them, and Sections 4 and 5 show how to build and run an application with Hadoop MapReduce, and connect to HBase.

2   Installing Hadoop MapReduce and HDFS

This section presents the steps you need to do to install MapReduce and HDFS.

1. Download and install Java and Hadoop MarReduce. Read more about the compatible versions here: https://cwiki.apache.org/confluence/display/HADOOP/Hadoop+Java+Versions

2. Set the following environment variables.

```
export JAVA_HOME="/path/to/the/java/folder"
export HADOOP_HOME="/path/to/the/hadoop/folde"
export HADOOP_CONFIG="$HADOOP_HOME/etc/hadoop"
export PATH=$JAVA_HOME/bin:$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$PATH
```

3. Set `JAVA_HOME` in the `$HADOOP_CONFIG/hadoop-env.sh` shell script.

```
export JAVA_HOME="/path/to/the/java/folder"
```

4. Make two folders on local file system, where HDFS *namenode* and *datanode* store their data.

```
mkdir -p $HADOOP_HOME/hdfs/namenode
mkdir -p $HADOOP_HOME/hdfs/datanode
```

5. The `$HADOOP_CONFIG/core-site.xml` file contains information such as the URI of the namenode (master), the port number used for Hadoop instance, memory allocated for file system, memory limit for storing data, and the size of read/write buffers. Open `$HADOOP_CONFIG/core-site.xml` and add the following lines.

```
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://127.0.0.1:9000</value>
    <description>NameNode URI</description>
  </property>
</configuration>
```

6. The `$HADOOP_CONFIG/hdfs-site.xml` file contains information such as the value of replication data, as well as the namenode and datanode paths on your local file system. Here, you should replace the `PATH_TO_NAMENODE` and `PATH_TO_DATANODE` with the paths you defined in step 4.

```
<configuration>
  <property>
    <name>dfs.replication</name >
    <value>1</value>
  </property>

  <property>
    <name>dfs.namenode.name.dir</name>
    <value>file://PATH_TO_NAMENODE</value>
    <description>Path on the local filesystem ...</description>
  </property>

  <property>
    <name>dfs.datanode.data.dir</name>
    <value>file://PATH_TO_DATANODE</value>
    <description>Comma separated list of paths on the local filesystem ...</description>
  </property>
</configuration>
```

7. Format the namenode directory (DO THIS ONLY ONCE, THE FIRST TIME).

```
$HADOOP_HOME/bin/hdfs namenode -format
```

8. Start the namenode and datanode daemons.

```
$HADOOP_HOME/sbin/hadoop-daemon.sh start namenode
$HADOOP_HOME/sbin/hadoop-daemon.sh start datanode
```

If all the above steps are done correctly, you should see the running processing. To do so, execute the `jps` in a terminal to print out the HDFS running processes. You can also monitor the process through their web interfaces on the following addresses `http://127.0.0.1:50070` for the namenode and `http://127.0.0.1:50075` for the datanode. Now, let's try some HDFS commands:

- Create a new directory `/kth` on HDFS.

```
$HADOOP_HOME/bin/hdfs dfs -mkdir /kth
```

- Create a file, call it `big`, on your local filesystem and upload it to HDFS under `/sics`.

```
touch big
$HADOOP_HOME/bin/hdfs dfs -put big /kth
```

- View the content of `/kth` directory.

```
$HADOOP_HOME/bin/hdfs dfs -ls /kth
```

- Determine the size of file `big` on HDFS.

```
$HADOOP_HOME/bin/hdfs dfs -du -h /kth/big
```

- Print the first 5 lines of the file `big` to screen (the `big` file is empty, so you can add some lines of text to it before uploading it on the HDFS).

```
$HADOOP_HOME/bin/hdfs dfs -cat /kth/big | head -n 5
```

- Make a copy of the file `big` on HDFS, and call it `big_hdfscopy`

```
$HADOOP_HOME/bin/hdfs dfs -cp /kth/big /kth/big_hdfscopy
```

- Copy the file `big` to the local filesystem and name it `big_localcopy`

```
$HADOOP_HOME/bin/hdfs dfs -get /kth/big big_localcopy
```

- Check the entire HDFS filesystem for inconsistencies/problems

```
$HADOOP_HOME/bin/hdfs fsck /
```

- Delete the file `big` from HDFS

```
$HADOOP_HOME/bin/hdfs dfs -rm /kth/big
```

- Delete the folder `/kth` from HDFS

```
$HADOOP_HOME/bin/hdfs dfs -rm -r /sics
```

3   Installing HBase

Here, we explain how to install HBase in *pseudo-distributed* mode, where all daemons run on a single node

1. Install the SSH-server.

```
sudo apt install openssh-server
```

2. Download the Apache HBase from the following link
   https://hbase.apache.org/downloads.html

3. Set the following environment variables.

```
export HBASE_HOME="/path/to/the/hbase/folder"
export HBASE_CONF="$HBASE_HOME/conf"
export PATH=$HBASE_HOME/bin:$PATH
```

4. Edit JAVA_HOME in $HBASE_CONF/hbase-env.sh.

```
export JAVA_HOME="/path/to/the/java/folder"
```

5. Make a folder on local file system, where zookeeper stores its data.

```
mkdir -p $HBASE_HOME/zookeeper
```

6. Edit $HBASE_CONF/hbase-site.xml by adding the following lines. Replace the PATH_TO_ZOOKEEPER
   with the path you built in the step 4.

```
<configuration>
   <property>
      <name>hbase.rootdir</name>
      <value>hdfs://localhost:9000/hbase</value>
   </property>

   <property>
      <name>hbase.zookeeper.property.dataDir</name>
      <value>PATH_TO_ZOOKEEPER</value>
   </property>

   <property>
     <name>hbase.cluster.distributed</name>
     <value>true</value>
   </property>
</configuration>
```

7. Start HBase with the following command. Before starting it, make sure that Hadoop namenode and datanodes are running.

```
$HBASE_HOME/bin/start-hbase.sh
```

8. HBase creates its directory in HDFS. To see the created directory type the following command.

```
$HADOOP_HOME/bin/hdfs dfs -ls /hbase
```

9. Start and stop a backup HBase Master server (OPTIONAL ON PRODUCTION ENVIRONMENTS). This step is offered for testing and learning purposes only, because running multiple HMaster instances on the same hardware does not make sense in a production environment (in the same way that running a pseudo-distributed cluster does not make sense for production). The HMaster server controls the HBase cluster. You can start up to nine backup HMaster servers, which makes 10 total HMasters, counting the primary. To start a backup HMaster, use the $HBASE_HOME/bin/local-master-backup.sh command. For each backup master you want to start, add a parameter representing the port offset for that master. Each HMaster uses three ports (16010, 16020, and 16030 by default). The port offset is added to these ports, so using an offset of 2, the backup HMaster would use ports 16012, 16022, and 16032. The following command starts 3 backup servers using ports 16012/16022/16032, 16013/16023/16033, and 16015/16025/16035.

```
$HBASE_HOME/bin/local-master-backup.sh 2 3 5
```

10. To kill a backup master without killing the entire cluster, you need to find its process ID (PID). The PID is stored in a file with a name like /tmp/hbase-USER-X-master.pid. The only content of the file is the PID. You can use the kill -9 command to kill that PID. The following command will kill the master with port offset 1, but leave the cluster running:

```
cat /tmp/hbase-amir-1-master.pid | xargs kill -9
```

11. Start and stop additional RegionServers (OPTIONAL ON PRODUCTION ENVIRONMENTS). The HRegionServer manages the data in its StoreFiles as directed by the HMaster. Generally, one HRegionServer runs per node in the cluster. Running multiple HRegionServers on the same system can be useful for testing in pseudo-distributed mode. The $HBASE_HOME/bin/local-regionservers.sh command allows you to run multiple RegionServers. It works in a similar way to the local-master-backup.sh command, in that each parameter you provide represents the port offset for an instance. Each RegionServer requires two ports, and the default ports are 16020 and 16030. However, the base ports for additional RegionServers are not the default ports since the default ports are used by the HMaster, which is also a RegionServer since HBase version 1.0.0. The base ports are 16200 and 16300 instead. You can run 99 additional RegionServers that are not a HMaster or backup HMaster, on a server. The following command starts four additional RegionServers, running on sequential ports starting at 16202/16302 (base ports 16200/16300 plus 2).

```
$HBASE_HOME/bin/local-regionservers.sh start 2 3 4 5
```

12. To stop a RegionServer manually, use the $HBASE_HOME/bin/local-regionservers.sh command with the stop parameter and the offset of the server to stop.

```
$HBASE_HOME/bin/local-regionservers.sh stop 3
```

Now, let's go through the following steps to create a table in HBase and test it.

- Start the HBase shell

```
$HBASE_HOME/bin/hbase shell
```

- Create a table called `test` with the column family `cf`

```
# ' is a single quotation
create 'test', 'cf'
```

- Use the command `describe` to get the description of the table

```
describe 'test'
```

- Print the information about your table

```
list 'test'
```

- Put data into your table (the first insert is at `row1`, column `cf:a`, with a value of `value1`). Columns in HBase are comprised of a column family prefix, e.g., `cf`, followed by a colon and then a column qualifier suffix, e.g., `a`.

```
put 'test', 'row1', 'cf:a', 'value1'
put 'test', 'row2', 'cf:b', 'value2'
put 'test', 'row3', 'cf:c', 'value3'
```

- Scan the table for all data at once

```
scan 'test'
```

- To get a single row of data at a time, use the `get` command

```
get 'test', 'row1'
```

- If you want to delete a table or change its settings, as well as in some other situations, you need to disable the table first, using the `disable` command. You can re-enable it using the `enable` command

```
disable 'test'
enable 'test'
```

- To delete a table, use the `drop` command.

```
disable 'test'
drop 'test'
```

- Exit the HBase shell

```
exit
```

4 Word Count in MapReduce

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.

4.1 The Mapper Class

Here is the word count mapper class:

```java
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 or the `map` method, i.e., `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 would be a long value assigned in default based on the position of `Text` in the input file. Our output from the mapper is of the format (word, 1) hence the data type 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 and value to the mapper. 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 its 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.

4.2 The Reducer Class

Here is the word count reducer class:

```java
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 the data type of input key and value to the reducer and the last two refer to the data type of output key and value. Our mapper emits output as (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>`.

6

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 the 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.

4.3 The 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. Bellow you see the complete code of the word count. Here, we assume there are two input files, `file0` and `file1`, uploaded on HDFS, and our code reads those files and counts their words. Then, we should go through the following steps to compile and run the code:

1. Start the HDFS namenode and datanode (if they are not running). Then create a folder `input` in HDFS, and upload the files in it.

```
$HADOOP_HOME/sbin/hadoop-daemon.sh start namenode
$HADOOP_HOME/sbin/hadoop-daemon.sh start datanode

cd lab1/src/wordcount
$HADOOP_HOME/bin/hdfs dfs -mkdir -p input
$HADOOP_HOME/bin/hdfs dfs -put data/file0 input/file0
$HADOOP_HOME/bin/hdfs dfs -put data/file1 input/file1
$HADOOP_HOME/bin/hdfs dfs -ls input
```

2. Set the `HADOOP_CLASSPATH` environment variable.

```
export HADOOP_CLASSPATH=$($HADOOP_HOME/bin/hadoop classpath)
```

3. Change directory to the `src` folder and make a target directory, `wordcount_classes`, to keep the compiled files. Then, compile the code and make a final jar file.

```
cd lab1/src

mkdir wordcount_classes

javac -cp $HADOOP_CLASSPATH -d wordcount_classes wordcount/WordCount.java

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

4. Run the application.

```
$HADOOP_HOME/bin/hadoop jar wordcount.jar wordcount.WordCount input output
```

5. Check the output in HDFS.

```
$HADOOP_HOME/bin/hdfs dfs -ls output
$HADOOP_HOME/bin/hdfs dfs -cat output/part-r-00000
```

```java
package wordcount;

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);
  }
}
```

5   MapReduce and HBase

Here, we show a sample MapReduce program that reads data from the HBase table, does some aggregation and writes the output to another HBase table.

5.1   The Mapper Class

Here is the mapper class:

```java
public static class hbaseMapper extends TableMapper<Text, IntWritable> {
  public void map(ImmutableBytesWritable rowKey, Result columns, Context context)
              throws IOException, InterruptedException {
    try {
        String inKey = new String(rowKey.get());

        String oKey = inKey.split("#")[0];

        byte[] bSales = columns.getValue(Bytes.toBytes("cf"), Bytes.toBytes("sales"));
        String sSales = new String(bSales);
        Integer sales = new Integer(sSales);

        context.write(new Text(oKey), new IntWritable(sales));
    } catch (RuntimeException e) {
      e.printStackTrace();
    }
  }
}
```

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

1. Get the `rowKey` and convert it to `String`.

2. Set new key having only date.

3. Get sales column in byte format first and then convert it to string (as it is stored as string from hbase shell).

4. Emit date and sales values

5.2   The Reducer Class

Here is the reducer class:

```java
public static class hbaseReducer extends TableReducer<Text, IntWritable, ImmutableBytesWritable> {
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
                throws IOException, InterruptedException {
    try {
        int sum = 0;

        for (IntWritable sales : values) {
          Integer intSales = new Integer(sales.toString());
              sum += intSales;
        }

        Put insHBase = new Put(key.getBytes());

        insHBase.addColumn(Bytes.toBytes("cf"), Bytes.toBytes("sum"), Bytes.toBytes(sum));

        context.write(null, insHBase);
    } catch (Exception e) {
        e.printStackTrace();
      }
  }
}
```

The functionality of the **reducer** method is as follows:

1. Initialize the variable `sum` as 0.

2. Loop through different sales values and add it to `sum`.

3. Create HBase put `insHBase` with rowkey as date.

4. Insert `sum` value to HBase.

5. Write data to Hbase table

5.3 The Driver Class

Below, you can see the complete code, including the driver class, as well as how to build a table in HBase and compile and run the code. Here, we assume there are two HBase tables, named `test1` and `test2`, and our code reads a column from `test1`, aggregates the values and writes the result in `test2`.

1. Start the HBase and the HBase shell.

```
$HBASE_HOME/bin/start-hbase.sh
$HBASE_HOME/bin/hbase shell
```

2. Create the HBase tables `test1` and `test2`. For the table `test1`, the rowkey is composed of `date` and `store number` and we have `sales` value for each store. We shall do an aggregation on `date` to get total `sales`.

```
create 'test1', 'cf'

put 'test1', '20130101#1', 'cf:sales', '100'
put 'test1', '20130101#2', 'cf:sales', '110'
put 'test1', '20130102#1', 'cf:sales', '200'
put 'test1', '20130102#2', 'cf:sales', '210'

create 'test2', 'cf'

scan 'test1'
scan 'test2'
```

3. Set the `HADOOP_CLASSPATH` environment variable.

```
export HADOOP_CLASSPATH=$($HADOOP_HOME/bin/hadoop classpath)
export HBASE_CLASSPATH=$($HBASE_HOME/bin/hbase classpath)
export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:$HBASE_CLASSPATH
```

4. Change directory to the `src` folder and make a target directory, `hbase_classes`, to keep the compiled files. Then, compile the code and make a final jar file.

```
cd lab1/src

mkdir hbase_classes

javac -cp $HADOOP_CLASSPATH -d hbase_classes hbase/HBaseMapReduce.java

jar -cvf hbaseMapreduce.jar -C hbase_classes/ .
```

5. Run the application.

```
$HADOOP_HOME/bin/hadoop jar hbaseMapreduce.jar hbase.HBaseMapReduce
```

6. Check the output in HBase. The sum values are displayed as bytes (HBase stores everything as bytes), we can convert it to readable integer format in HBase shell.

```
scan 'test2'
org.apache.hadoop.hbase.util.Bytes.toInt("\x00\x00\x00\xD2".to_java_bytes)
org.apache.hadoop.hbase.util.Bytes.toInt("\x00\x00\x01\x9A".to_java_bytes)
```

```java
package hbase;

import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.mapreduce.TableMapper;
import org.apache.hadoop.hbase.mapreduce.TableReducer;
import org.apache.hadoop.hbase.mapreduce.TableMapReduceUtil;
import org.apache.hadoop.hbase.io.ImmutableBytesWritable;
import org.apache.hadoop.hbase.filter.FirstKeyOnlyFilter;

public class HBaseMapReduce {
  public static class hbaseMapper extends TableMapper<Text, IntWritable> {
    public void map(ImmutableBytesWritable rowKey, Result columns, Context context)
                   throws IOException, InterruptedException {
      try {
            String inKey = new String(rowKey.get());

            String oKey = inKey.split("#")[0];

            byte[] bSales = columns.getValue(Bytes.toBytes("cf"), Bytes.toBytes("sales"));
            String sSales = new String(bSales);
            Integer sales = new Integer(sSales);

            context.write(new Text(oKey), new IntWritable(sales));
          } catch (RuntimeException e) {
            e.printStackTrace();
      }
    }
  }

  public static class hbaseReducer extends TableReducer<Text, IntWritable, ImmutableBytesWritable> {
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
                     throws IOException, InterruptedException {
      try {
            int sum = 0;

            for (IntWritable sales : values) {
              Integer intSales = new Integer(sales.toString());
                  sum += intSales;
            }

            Put insHBase = new Put(key.getBytes());

                insHBase.addColumn(Bytes.toBytes("cf"), Bytes.toBytes("sum"), Bytes.toBytes(sum));

            context.write(null, insHBase);
      } catch (Exception e) {
            e.printStackTrace();
        }
    }
  }
}
```

11

```java
  public static void main(String[] args) throws Exception {
    Configuration conf = HBaseConfiguration.create();

    // define scan and define column families to scan
    Scan scan = new Scan();
    scan.addFamily(Bytes.toBytes("cf"));
    Job job = Job.getInstance(conf);
    job.setJarByClass(HBaseMapReduce.class);

    // define input hbase table
    TableMapReduceUtil.initTableMapperJob("test1", scan, hbaseMapper.class, Text.class, IntWritable.class, job);

    // define output table
    TableMapReduceUtil.initTableReducerJob("test2", hbaseReducer.class, job);
    job.waitForCompletion(true);
  }
}
```