Apache Hadoop – Introduction

Hadoop Teaching Cluster

General Notes

Homework for this course is completed using the Hadoop Teaching Cluster (VMs) which runs the CentOS 7 Linux distribution. This Hadoop Cluster runs on a Hortonworks Data Platform 2.6.5 installation (https://de.hortonworks.com/products/data-platforms/hdp/). You can login to the Cluster via Web UI (http://bda-job.et-it.hs-offenburg.de:8080) and via SSH (Putty).

Getting Started

Every student has own login credentials, which are **studentX** with password **StudentX**.

Working with the Console

In some command-line steps in the labs, you will see lines like this:

```
$ hdfs dfs –put shakespeare /user/studentX/shakespeare
```

The dollar sign ($) at the beginning of each line indicates the Linux shell prompt. The actual prompt will include additional information (`[studentX@localhost workspace]$`)

but this is omitted from these instructions for brevity.

The backslash (\) at the end of the first line signifies that the command is not completed, and continues on the next line. You can enter the code exactly as shown (on two lines), or you can enter it on a single line. If you do the latter, you should *not* type in the backslash.

First Login

Try to login with your credentials:

a) Ambari UI by opening http://bda-job.et-it.hs-offenburg.de:8080
b) Putty by adding a **SSH** connection to the server (=Host Name) **bda-job.et-it.hs-offenburg.de**, Port **22**
    a. **Putty Download: https://www.putty.org**

File Preparation

The files needed for this course can be found in the directory /etc/training_materials and /etc/wordcount_new. Please copy these directories into your home directory, before continuing with the next steps. Use these commands:

```
$ cp -r /etc/training_materials/  ~/training_materials
$ cp -r /etc/wordcount_new/  ~/wordcount
```

Lab: Using HDFS

Files Used in This Exercise:
~/training_materials/developer/data/shakespeare.tar.gz
~/training_materials/developer/data/access_log.gz

In this lab you will begin to get acquainted with the Hadoop tools. You will manipulate files in HDFS, the Hadoop Distributed File System.

Hadoop

Hadoop is already installed, configured, and running on the Cluster. Most of your interaction with the system will be through command-line wrappers called `hadoop` and `hdfs`. If you run these programs with no arguments, it prints a help message. To try this, run the following commands in a terminal window:

```
$ hadoop
$ hdfs
```

The `hadoop` command is subdivided into several subsystems. For example, there is a subsystem for working with files in HDFS (deprecated) and another for launching and managing MapReduce processing jobs.
Instead of using the `hadoop fs` command you should use the new `hdfs dfs` command.

Step 1: Exploring HDFS

The subsystem associated with HDFS in the Hadoop wrapper program is called `FsShell`. This subsystem can be invoked with the command `hdfs dfs`.
Open a SSH session (if one is not already open) by using the Putty client. In the terminal window, enter:

```
$ hdfs dfs
```

You see a help message describing all the commands associated with the `FsShell` subsystem.

Enter: `$ hdfs dfs –ls /`

This shows you the contents of the root directory in HDFS. There will be multiple entries, one of which is `/user`. Individual users have a "home" directory under this directory, named after their username; your username in this course is `studentX`, therefore your home directory is `/user/studentX`.

Try viewing the contents of the `/user` directory by running:

```
$ hdfs dfs –ls /user
```

You will see your home directory in the directory listing.
List the contents of your home directory by running:

```
$ hdfs dfs –ls /user/studentX
```

There are no files yet, so the command silently exits. This is different from running `hdfs dfs –ls /foo`, which refers to a directory that doesn't exist. In this case, an error message would be displayed.

Note that the directory structure in HDFS has nothing to do with the directory structure of the local file system; they are completely separate namespaces.

Step 2: Uploading Files

Besides browsing the existing file system, another important thing you can do with `FsShell` is to upload new data into HDFS.

1. Change directories to the local file system directory containing the sample data we will be using in the homework labs.

   ```
   $ cd ~/training_materials/developer/data
   ```

   If you perform a regular Linux `ls` command in this directory, you will see a few files, including two named `shakespeare.tar.gz` and `shakespeare-stream.tar.gz`. Both of these contain the complete works of Shakespeare in text format, but with different formats and organizations. For now we will work with **`shakespeare.tar.gz`**.

2. Unzip `shakespeare.tar.gz` by running:

   ```
   $ tar zxvf shakespeare.tar.gz
   ```

   This creates a directory named `shakespeare/` containing several files on your local file system.

3. Upload this directory into HDFS:

   ```
   $ hdfs dfs -put shakespeare /user/studentX/shakespeare
   ```

   This copies the local `shakespeare` directory and its contents into a remote HDFS directory named `/user/studentX/shakespeare`.

4. List the contents of your HDFS home directory now:

   ```
   $ hdfs dfs -ls /user/studentX
   ```

   You should see an entry for the `shakespeare` directory.

5. Now try the same `dfs -ls` command but without a path argument:

   ```
   $ hdfs dfs -ls
   ```

   You should see the same results. If you don't pass a directory name to the `-ls` command, it assumes you mean your home directory, i.e. `/user/studentX`.

   Hint: Relative paths

   If you pass any relative (non-absolute) paths to FsShell commands (or use relative paths in MapReduce programs), they are considered relative to your home directory.

6. We will also need a sample web server log file, which we will put into HDFS for use in future labs. This file is currently compressed using GZip. Rather than extract the file to the local disk and then upload it, we will extract and upload in one step. First, create a directory in HDFS in which to store it:

   ```
   $ hdfs dfs -mkdir weblog
   ```

7. Now, extract and upload the file in one step. The `-c` option to `gunzip` uncompresses to standard output, and the dash (`-`) in the `hdfs dfs -put` command takes whatever is being sent to its standard input and places that data in HDFS.

   ```
   $ gunzip -c access_log.gz | hdfs dfs -put - weblog/access_log
   ```

8. Run the `hdfs dfs –ls` command to verify that the log file is in your HDFS home directory.
9. The access log file is quite large – around 500 MB. Create a smaller version of this file, consisting only of its first 5000 lines, and store the smaller version in HDFS. You can use the smaller version for testing in subsequent labs.

```
$ hdfs dfs –mkdir testlog
$ gunzip -c access_log.gz | head –n 5000 \
      | hdfs dfs -put - testlog/test_access_log
```

Step 3: Viewing and Manipulating Files

Now let's view some of the data you just copied into HDFS.

1. Enter: `$ hdfs dfs –ls shakespeare`

   This lists the contents of the `/user/studentX/shakespeare` HDFS directory, which consists of the files `comedies`, `glossary`, `histories`, `poems`, and `tragedies`.
2. The `glossary` file included in the compressed file you began with is not strictly a work of Shakespeare, so let's remove it:

   ```
   $ hdfs dfs –rm shakespeare/glossary
   ```

   Note that you *could* leave this file in place if you so wished. If you did, then it would be included in subsequent computations across the works of Shakespeare, and would skew your results slightly. As with many real-world big data problems, you make trade-offs between the labor to purify your input data and the precision of your results.
3. Enter: `$ hdfs dfs –cat shakespeare/histories | tail –n 50`

   This prints the last 50 lines of *Henry IV, Part 1* to your terminal. This command is handy for viewing the output of MapReduce programs. Very often, an individual output file of a MapReduce program is very large, making it inconvenient to view the entire file in the terminal. For this reason, it's often a good idea to pipe the output of the `dfs –cat` command into `head`, `tail`, `more`, or `less`.

4. To download a file to work with on the local file system use the `dfs –get` command. This command takes two arguments: an HDFS path and a local path. It copies the HDFS contents into the local file system:

   ```
   $ hdfs dfs –get shakespeare/poems ~/shakepoems.txt
   $ less ~/shakepoems.txt
   ```

Other Commands

There are several other operations available with the `hdfs dfs` command to perform most common file system manipulations: `mv`, `cp`, `mkdir`, etc.

Enter: `$ hdfs dfs`

This displays a brief usage report of the commands available within `FsShell`. Try playing around with a few of these commands if you like.

Lab: Running a MapReduce Job

Files and Directories Used in this Exercise
Source directory: ~/ wordcount/src/stubs
Files:

1. WordCount.java: A simple MapReduce driver class.
2. WordMapper.java: A mapper class for the job.
3. SumReducer.java: A reducer class for the job.
4. wc.jar: The compiled, assembled WordCount program

In this lab you will compile Java files, create a JAR, and run MapReduce jobs.
In addition to manipulating files in HDFS, the wrapper program `hadoop` is used to launch MapReduce jobs. The code for a job is contained in a compiled JAR file. Hadoop loads the JAR into HDFS and distributes it to the worker nodes, where the individual tasks of the MapReduce job are executed. One simple example of a MapReduce job is to count the number of occurrences of each word in a file or set of files. In this lab you will compile and submit a MapReduce job to count the number of occurrences of every word in the works of Shakespeare.

Compiling and Submitting a MapReduce Job

1. In a terminal window, change to the lab source directory, and list the contents:

```
$ cd ~/wordcount/src
$ ls
```

List the files in the `stubs` package directory: `$ ls stubs`

The package contains the following Java files:
`WordCount.java`: A simple MapReduce driver class.
`WordMapper.java`: A mapper class for the job.
`SumReducer.java`: A reducer class for the job.

Examine these files if you wish, but do not change them. Remain in this directory while you execute the following commands.

2. Before compiling, examine the classpath Hadoop is configured to use:

```
$ hadoop classpath
```

This shows lists the locations where the Hadoop core API classes are installed.

3. Compile the three Java classes:

```
$ javac -classpath `hadoop classpath` stubs/*.java
```

In the command above, the quotes around `hadoop classpath` are backquotes. This runs the `hadoop classpath` command and uses its output as part of the `javac` command. The compiled (`.class`) files are placed in the `stubs` directory.

Since you added the necessary libraries by the `-classpath` option in your `javac` command, the

project can be compiled, as the compiler can find all needed files in those paths.

An alternative to compile the project with the command line `javac` command is to create a project in your Java IDE and add the dependencies there (e.g. by using Maven). We will look at such a project structure in the next lab.

4. Collect your compiled Java files into a JAR file:

```
$ jar cvf wc.jar stubs/*.class
```

5. Submit a MapReduce job to Hadoop using your JAR file to count the occurrences of each word in Shakespeare. First set the HADOOP_CLASSPATH to your JAR file that you created before.

```
$ export HADOOP_CLASSPATH=/home/studentX/wordcount/src/wc.jar
```

```
$ hadoop jar wc.jar stubs.WordCount shakespeare wordcounts
```

This `hadoop jar` command names the JAR file to use (`wc.jar`), the class whose `main` method should be invoked (`stubs.WordCount`), and the HDFS input and output directories to use for the MapReduce job. Your job reads all the files in your HDFS `shakespeare` directory, and places its output in a new HDFS directory called `wordcounts`.

6. Try running this same command again without any change:

```
$ hadoop jar wc.jar stubs.WordCount shakespeare wordcounts
```

Your job halts right away with an exception, because Hadoop automatically fails if your job tries to write its output into an existing directory. This is by design; since the result of a MapReduce job may be expensive to reproduce, Hadoop prevents you from accidentally overwriting previously existing files.

7. Review the result of your MapReduce job:

```
$ hdfs dfs –ls wordcounts
```

This lists the output files for your job. (Your job ran with only one Reducer, so there should be one file, named `part-r-00000`, along with a `_SUCCESS` file).

8. View the contents of the output for your job:

```
$ hdfs dfs –cat wordcounts/part-r-00000 | less
```

You can page through a few screens to see words and their frequencies in the works of Shakespeare. (The spacebar will scroll the output by one screen; the letter '`q`' will quit the `less` utility.) Note that you could have specified `wordcounts/*` just as well in this command.

Wildcards in HDFS file paths

Take care when using wildcards (e.g. `*`) when specifying HFDS filenames; because of how Linux works, the shell will attempt to expand the wildcard before invoking `hdfs` / `hadoop`, and then pass incorrect references to local files instead of HDFS files. You can prevent this by enclosing the wildcarded HDFS filenames in single quotes, e.g. `hdfs dfs –cat 'wordcounts/*'`

9. Try running the WordCount job against a single file:

```
$ hadoop jar wc.jar stubs.WordCount shakespeare/poems pwords
```

When the job completes, inspect the contents of the `pwords` HDFS directory.

10. Clean up the output files produced by your job runs:

```
$ hdfs dfs –rm –r wordcounts pwords
```

Trash mechanism in HDFS

When you delete files by `hdfs dfs -rm` command, the files are moved to the trash directory first. It will hold the data for 6 hours (set in Ambari Administration), until it's deleted completely.

The directory of your user's trash is **`/user/studentX/.Trash`**

If you want to delete files directly, without moving them to Trash first, use the rm command with the -skipTrash option, e.g.

```
hdfs dfs –rm –r -skipTrash wordcounts pwords
```

Stopping MapReduce Jobs

It is important to be able to stop jobs that are already running. This is useful if, for example, you accidentally introduced an infinite loop into your Mapper.

An important point to remember is that pressing ^C to kill the current process (which is displaying the MapReduce job's progress) does not actually stop the job itself.

A MapReduce job, once submitted to Hadoop, runs independently of the initiating process, so losing the connection to the initiating process does not kill the job. Instead, you need to tell the Hadoop JobTracker to stop the job.

1. Copy the wordcount_infinite project into your home, compile it and build a jar.

```
$ cp -r /etc/wordcount_infinite ~/wordcount_infinite
$ cd ~/wordcount_infinite/src
$ javac -classpath `hadoop classpath` stubs/*.java
$ jar cvf wc2.jar stubs/*.class
```

2. Start another word count job like you did in the previous section:

```
$ hadoop jar wc2.jar stubs.WordCount shakespeare output_infinite
```

3. While this job is running, open another terminal window and enter:

```
$ mapred job -list
```

This lists the job ids of all running jobs. A job id looks something like:

```
job_200902131742_0002
```

4. Copy the job id, and then kill the running job by entering:

```
$ mapred job -kill <jobid>
```

The JobTracker kills the job, and the program running in the original terminal completes.