Cluster Warmup

This document provides information on how to take your first steps in the class cluster for Project 1.

Logging into the cluster

To get access to the cluster, please check the `computing-cluster` document for points on how to generate an SSH key and get an account on the cluster. After your account is generated, you will be able to log into the cluster's main virtual machine running `ssh`:

```
$ ssh -p 4422 -i "caminho/para/chave" usuario@150.164.203.31
[vm:~]%
```

The command above van be simplified by creating a `Host` on your `.ssh/config` file (adjust your username to match your ID). With this configuration, you can use `cloudvm` directly instead of specifying the user, host, and port every time.

```
Host cloudvm
    HostName 150.164.203.31
    Port 4422
    User italocunha
```

To copy a file to the cluster, use the `scp` command, it receives as parameters the source and destination files in the following format:

```
scp <localFile> <user>@<remoteHost>:<remotepath>/<remotefile>
```

Here is an example copying a file called `README.md` to the cluster:

```
scp README.md cloudvm:remoteDirectory/README.md
```

Setting up the environment

The main VM is set up to automatically configure the user environment to access the tools and frameworks we will use in the assignments (for example, Spark, HDFS, Kubernetes). If the commands in the following sections do not work, we may need to manually set up the environment.

The environment is loaded on user log-in from `/etc/profile.d`, but if your shell does not load

the default environment (for example, because you've changed it from `bash` to `zsh`), then running the following commands will load the environment:

```
source /etc/profile.d/hadoop.sh
source /etc/profile.d/spark.sh
```

You should consider adding these commands to your shell's initialization scripts.

A look at these files will show you that they set several environment variables (like `HADOOP_HOME`), which are used by the Hadoop and Spark scripts to locate other files in the installation. They also set the important `PATH` variable, which is where your shell looks for commands. With the `PATH` variable correctly set up, the commands in the following section should work.

HDFS

HDFS's interface resembles that of a local file system. We call the `hdfs` binary, and then the `dfs` module (Distributed File System), and then use several subcommands of the `dfs` module to operate on the file system.

The commands below will perform basic operations on HDFS. Remember that your files should be under the `/user/<login>` directory in HDFS; replace `cunha` in the commands below with your username on the cluster:

```
# Create a directory test in the user's directory:
hdfs dfs -mkdir /user/cunha/test

# Adding a file to the distributed filesystem:
hdfs dfs -put $HADOOP_HOME/LICENSE.txt /user/cunha/test

# Listing the contents of a directory:
hdfs dfs -ls /user/cunha/test

# Determine the size of a file on HDFS:
hdfs dfs -du -h /user/cunha/test

# Get the contents of a file, then pike into the `head` command
# to get only the first 10 lines:
hdfs dfs -cat /user/cunha/test/LICENSE.txt | head

# Making a copy of a file:
hdfs dfs -cp /user/cunha/test/LICENSE.txt /user/cunha/test/LICENSE.backup

# Getting a local copy of a file on HDFS:
hdfs dfs -get /user/cunha/test/LICENSE.txt LICENSE.local

# Check the integrity of the filesystem:
hdfs fsck /

# Delete a file on HDFS:
```

```
hdfs dfs -rm /user/cunha/test/LICENSE.backup

# Recursively remove a directory on HDFS:
hdfs dfs -rm -r /user/cunha/test
```

Spark

Spark can be accessed interactively, similar to how we can run `python3` or `ipython` in a terminal to test commands. To get an interactive shell, we use either the `spark-shell` or `pyspark` commands when using Scala or Python, respectively.

As a warm-up exercise, let's create a file containing "Hello World!" on HDFS, then read this file from inside Spark. (Remember to replace `cunha` with your username.)

```
echo "Hello World" > hello.txt
hdfs dfs -put hello.txt /user/cunha/
```

Please inform the number of *executors* (the number of concurrent execution threads) passing `--num-executors`, the number of CPU cores used by the executors passing `--executor-cores`, and the amount of memory to be used by executors `--executor-memory`. Here is an example invocation of `pyspark`:

```
$ pyspark --num-executors 2 \
        --executor-cores 2 \
        --executor-memory 1024M
(...)
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.2.0
      /_/

Using Python version 3.8.10 (default, Nov 26 2021 20:14:08)
Spark context Web UI available at http://hostname:4040
Spark context available as 'sc'.
SparkSession available as 'spark'.
```

Note that `pyspark` already creates a SparkContext and a SparkSession, which we can use interactively. For example, let's read a file, get its lines, and then compute the length of each line by typing the following commands in the interactive shell:

```
rdd = sc.textFile("hdfs:/user/cunha/hello.txt")
lines = rdd.count()
rdd.collect()
rdd2 = rdd.map(len)
rdd2.collect()
```

Note that the `textFile` function returns an `RDD` (Resilient Distributed Dataset) object, which operates on HDFS data. The RDD interface is different from that of a normal file to ensure efficient access to the file's content. This is because HDFS must support multi-terabyte databases.

RDDs are created by distributing a list of items across HDFS blocks. RDDs can be read from HDFS (like above), but if you need to create an RDD by hand, then you can use the `SparkContext.parallelize` function. You could then, for example, save the returned RDD to a file:

```
outrdd = sc.parallelize([lines])
# The following will fail if the output directory exists:
outrdd.saveAsTextFile("hdfs:/user/cunha/hello-linecount")
```

You can later check the contents of the `hdfs:/user/cunha/hello-linecount` *directory*. The RDD is split into multiple *parts*, which can later be read in parallel. One of the parts will contain the number of lines in `hello.txt`:

```
$ hdfs dfs -cat /user/cunha/hello-linecount/part-00001
1
```

To submit a job for execution, for example, after finishing its implementation and testing, give preference to `spark-submit`. You have to specify dependencies and the entry point of your job; this is done differently depending on whether the application is written in Scala or Python. For our simple Python warm-up program, we have no dependencies and can just pass the program directly as the entry point. Details on submitting jobs to Spark is available here.

```
spark-submit --num-executors 2 \
        --executor-cores 2 \
        --executor-memory 1024M \
        spark-warmup.py
```

Because our application runs outside PySpark when we submit it using `spark-submit`, we need to manually create the SparkSession and get the SparkContext. Here is the code inside `spark-warmup.py` submitted in the previous command:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .appName("HelloLines") \
    .getOrCreate()
sc = spark.sparkContext
rdd = sc.textFile("hdfs:/user/cunha/hello.txt")
lines = rdd.count()
outrdd = sc.parallelize([lines])
# The following will fail if the output directory exists:
outrdd.saveAsTextFile("hdfs:/user/cunha/hello-linecount-submit")
```

```
sc.stop()
```

> Do not leave an interactive shell open when not using it. Interactive shells allocate resources that remain occupy CPU and RAM while the shell is open, occupying resources that could be used to run other tasks. Be particularly mindful of this close to assignment deadlines.

Jupyter Notebook

The cluster has Jupyter notebook installed for those who want to program in PySpark/Python. To use it, we suggest the following steps:

1. Start the Jupyter service on the cluster, choose a free port number (something random between 20000 and 55000 should work):

   ```
   jupyter notebook --no-browser --port=<remotePort>
   ```

2. Create an SSH tunnel on your machine to access Jupyter on the cluster from your machine. The following command will forward all data to a given `<localPort>` on your machine to a `<remotePort>` on the cluster's main VM. The `remotePort` must be the same one used above, but we suggest you also set `localPort` identical to `remotePort` to ease the next step.

   ```
   ssh -fNT -L <localPort>:localhost:<remotePort> \
           <username>@vcm-30384.vm.duke.edu
   ```

3. Access the URL provided by Jupyter's log on your browser. It should look something like this: `http://localhost:51515/?token=c1hugeHexadecimalStringGoesInHere26`. (I used 51515 as my `localPort` and `remotePort`, you should use different numbers or conflicts may occur.) From the Web interface, click the `New` button and then `Notebook`.

As Jupyter also runs outside the PySpark shell, we need to create the SparkSession and get the SparkContext manually, as we did in `spark-warmup.py` above. As Jupyter also does not get the command-line parameters we passed to `pyspark` and `spark-submit`, you must configure the resources (memory, cores, and executors) when initializing the SparkSession. The following will create a SparkSession with the same parameters used in `spark-submit` above:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .appName("HelloLines") \
    .config("spark.executor.instances", "2") \
    .config("spark.executor.cores", "2") \
    .config("spark.executor.memory", "1024M") \
    .getOrCreate()
```

```
sc = spark.sparkContext
```

Running Jupyter within VSCode

You can also run Jupyter kernel within VSCode. You will need Python and Jupyter extensions, and you need to install dependencies manually (e.g., `pyspark`) using `pip install`. Note that you will need to go on the extensions tab in VSCode and install the Python and Jupyter extensions on the virtual machine even if you already have them installed on your local machine. Finally, before creating a `SparkSession` instance, you will need to configure the environment used by the Jupyter kernel. (These variables are set up already in your shell by the initialization scripts in `/etc/profile.d`, but are not set up in VSCode.) Adding this on the first cells in the notebook should be enough:

```
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["HADOOP_INSTALL"] = "/home/hadoop/hadoop"
os.environ["HADOOP_HOME"] = os.environ["HADOOP_INSTALL"]
os.environ["HADOOP_MAPRED_HOME"] = os.environ["HADOOP_INSTALL"]
os.environ["HADOOP_COMMON_HOME"] = os.environ["HADOOP_INSTALL"]
os.environ["HADOOP_HDFS_HOME"] = os.environ["HADOOP_INSTALL"]
os.environ["HADOOP_YARN_HOME"] = os.environ["HADOOP_INSTALL"]
os.environ["HADOOP_CONF_DIR"] = os.path.join(os.environ["HADOOP_INSTALL"], "/etc/hadoop")
os.environ["SPARK_HOME"] = "/home/hadoop/spark"
sys.path.insert(0, os.path.join(os.environ["SPARK_HOME"], "python"))
sys.path.append(os.path.join(os.environ["SPARK_HOME"], "python/lib/py4j-0.10.9.2-src.zip"))
```

Spark Run Monitoring

When a Spark application is running, the user can access the logs generated by Spark, available in the Spark UI. `pyspark` prints the port number the UI is running on once it starts execution. You can make another SSH tunnel to access the Spark UI. The last line below shows the port number you should SSH tunnel into:

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.2.0
      /_/

Using Python version 3.8.10 (default, Nov 26 2021 20:14:08)
Spark context Web UI available at http://localhost:4040
```

If running Spark on Jupyter, you can get the URL (with the port) for the Spark UI accessing the `uiWebUrl` field in the SparkContext:

```
from pyspark.sql import SparkSession
```

```
spark = SparkSession.builder \
    .appName("HelloLines") \
    .config("spark.executor.instances", "2") \
    .config("spark.executor.cores", "2") \
    .config("spark.executor.memory", "1024M") \
    .getOrCreate()
sc = spark.sparkContext
print(sc.uiWebUrl)
```