Creating an EMR cluster to run spark interactively: Complete instructions: April 24, 2018

- Run all the terminal commands from Desktop/DataScience/AWS
- Create the cluster
    - Login to AWS https://murt.signin.aws.amazon.com/console
        - User001 login
    - Create Cluster
        - Go to Advanced options
            - Or clone murtcluster/murtspot
        - Software: Spark and Hadoop
        - Network: murtvpc
        - Hardware

Master node:	m4.large	4 vCores, 8 GiB
Core node:	m4.4xlarge	32 vCores, 64 GiB

          Provision the core node on-demand so that it is cheap.
        - Security
          EC2 key pair: user001
          Master and slave default security groups
          Additional Security Groups: murtsecuritygroup (I have changed the ssh rule from custom to My IP here, so that I am able to ssh to the instance immediately.)
          Security group has to allow for ssh from local IP. (Top of the document has more on this.)
- Once the cluster is created, note the public DNS of the master
    - DNS: ec2-18-219-168-68.us-east-2.compute.amazonaws.com
    - Note: the public DNS will change every time the instance stops and starts again
    - `$ ssh -i "user001.pem" hadoop@ec2-18-219-168-68.us-east-2.compute.amazonaws.com`
    - `$ sudo yum update`
        - if there's a warning
    - pyspark is already installed since the cluster was provisioned with spark. Typing pyspark will bring it up.
    - Install Anaconda
        - https://repo.continuum.io/archive/ has the latest

      ```
      $ wget http://repo.continuum.io/archive/Anaconda2-5.1.0-Linux-x86_64.sh
      $ bash Anaconda2-5.1.0-Linux-x86_64.sh
              Do you wish the installer to prepend the Anaconda2 install location
              to PATH in your /home/hadoop/.bashrc ? [yes|no]
              [no] >>> yes
      $ source .bashrc
      $ conda update --prefix /home/hadoop/anaconda2 anaconda
      $ conda clean --all
      ```

        - In some instances, I did the last two steps (update / clean) at least twice each. Getting the very last version of Anaconda should help here. It seems like the machine cannot install all updates due to space problems depending on the master node properties. Conda clean frees up space (as much as 2GB) by deleting tarballs and temporary files. After the second try, I got a message saying all packages are up to date. Anaconda is now installed and updated.

- Set up Jupyter notebook
  Anaconda installed Jupyter. We just need to set up the environment variables
    - $ nano ~/.bashrc
      Add the following
      ```
      export SPARK_HOME="/usr/lib/spark"
      export PATH="$SPARK_HOME:$PATH"
      export PATH=$PATH:$SPARK_HOME/bin
      export PYTHONPATH=$SPARK_HOME/python/:$PYTHONPATH
      export ANACONDA_ROOT=~/anaconda2
      export PYSPARK_DRIVER_PYTHON=$ANACONDA_ROOT/bin/jupyter
      export PYSPARK_DRIVER_PYTHON_OPTS='notebook' pyspark
      export PYSPARK_PYTHON=/usr/bin/python
      ```
        - In the past, I used
          export PYSPARK_PYTHON=$ANACONDA_ROOT/bin/python
          also. The location of default python may have changed (or something).

- Create a directory (cons) on the master.
    - $ mkdir cons
- Then, using another terminal, copy csv files there. I have a shell script that has the copy commands. The master DNS needs to change every time a cluster is created.
    - $ bash uploadtocluster.sh
- In the past I was able to read CSV files directly into spark but something may have changed. Now I have to copy those CSV files to HDFS first. In the master node terminal
    - $ cd cons
    - $ hadoop fs -put Products_recoded.csv /user/hadoop/Products_recoded.csv
    - $ hadoop fs -put Issues_recoded.csv /user/hadoop/Issues_recoded.csv
- Log off from the master, then log back on.
- From the command line, type
  $ pyspark
- This brings up an odd looking screen. There are three questions, Allow token (twice) the Q for quit which brings up the typical Notebook running with warnings screen. The screen also has a token, which will be needed.
- From another terminal, connect to the notebook:
  $ ssh -i "user001.pem" -L 8000:localhost:8888
  hadoop@ec2-18-219-168-68.us-east-2.compute.amazonaws.com
- From a web browser, go to http://localhost:8000/. Enter the token shown on the pyspark terminal. This should bring up the home directory in jupyter
- Run consumer complaints pyspark program.

Spark cluster information

- From the AWS Console, click on the cluster. On the Summary tab, click on Enable web connection.
- Follow the guides. There are two things: ssh tunnel and foxy proxy setup.

- $ ssh -i user001.pem -ND 8157 hadoop@ec2-18-219-168-68.us-east-2.compute.amazonaws.com
- The important thing is the port number. I used the default one (8157). The port number should go into foxy proxy setup.
- Select Spark History server, then click on the Environment tab.

Running the Spark program

- The program is done, but it'd make sense to make sure that the program is efficiently written (not a spark translation of a python program but a spark-from-scratch program that efficiently capitalizes on distributed nature of it.)
- Two constructs that could especially be improved are groupBy and filter.
- The big issue has been getting the final dataframe written into a CSV file. I finally got this done:

  ```
  $ df.write.option("header", "true").csv("compFinal")
  ```
  - This creates many (200) small CSV files in compFinal directory. They can then be merged using hadoop file tool inside the notebook:

    ```
    !hadoop fs -getmerge -nl compFinal ComplaintsFinal.csv
    ```
  - This creates a big, single CSV file. The file will have the header rows repeated so those rows will have to be filtered. But otherwise the CSV file got an R dataframe without a problem.
  - The issue before was trying to consolidate chunks of the dataframe into a single pandas dataframe or coalesce(1), which also consolidates the data in one executor. This write simply writes the chunks of the dataframe in HDFS.

Tuning Spark session parameters

- For the cluster I used a master (m4.large) and a core node (m4.4xlarge, 32 vCores, 64 GiB memory, spot at 15 cents an hour). This was way more than needed given that the data set is about 300 MB. But this allowed to me define executors as practice. (I also tried a version with a single machine (m4.4xlarge).)
  - I need to set aside 1 vCore and 1 GiB memory for OS / Hadoop deamons.
  - With 5 executor cores, I get 6 executors (6 * 5 = 30). I leave one aside for Applications manager and use the remaining 5 executors.
  - Memory per executor (64 - 1) / 5 = 10.5 GiB.
  - Leave about 7% for overhead: 1 GiB
  - Default parallelism is 2~4 times of

    spark.executor.instances * spark.executor.cores
    - https://rea.tech/how-we-optimize-apache-spark-apps/
  - I also explicitly set the deploy mode to cluster (spark driver lives on one of the core nodes)

```
confM = pyspark.SparkConf().setAll([('spark.executor.memory', '10g'), \
                                    ('spark.executor.cores', '5'), \
                                    ('spark.executor.instances', '25'), \
                                    ('spark.default.parallelism', '100'), \
                                    ('spark.yarn.executor.memoryOverhead', '1g'), \
                                    ('spark.submit.deployMode', 'cluster')])
```