**Stand Alone Mode (In a Single Cluster)**

Steps to run TallnWide in a single Cluster (Amazon AWS EMR Cluster)

1. Go to AWS Console and select EMR
2. Press Create Cluster and choose Advanced options
3. Under the options of EMR choose EMR-5.7.0, and select necessary softwares as mentioned in the paper
4. Click Next, choose the type and number of instance (m3.xlarge, 1 master and 7 slaves in our case)
5. Click Next, choose your .pem file to associate your account1
6. Create Cluster
7. Connect to the Master Node Using SSH and configure to View Web Interfaces Hosted on Amazon EMR Clusters using the following link: <https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-connect-master-node.html>
8. Install Apache Maven using the link: <https://maven.apache.org/install.html>
9. Run the following commands to execute TallnWide

$ git clone <https://github.com/DataMiningResearch/TallnWide.git>

$ cd TallnWide/TallnWide\_Accum\_Scheme/

$ mvn clean package

$ cp target/ PPCA-Tall-and-Wide-0.0.1-SNAPSHOT.jar stand\_alone/

$ cd stand\_alone/

$ spark-submit --master yarn --verbose --class org.qcri.sparkpca.StandAloneTallnWide --conf spark.driver.maxResultSize=0 --conf spark.network.timeout=4000s --driver-memory 10G --executor-memory 10G --driver-java-options "-Di=s3n://pragmatic-ppca/amazon.txt21176522x9874212.seq -Do=./ -DdataSet=amazon.txt21176522x9874212.seq -Drows=21176522 -Dcols=9874212 -DavgCols=20-Dpcs=10 -Drho=0 -Dtolerance=0.05 -DmaxIter=10" PPCA-Tall-and-Wide-0.0.1-SNAPSHOT.jar

**Steps of creating Geo Spark & Hadoop Cluster using spark-ec2 script**

**Author: Md. Mehrab Tanjim 29/01/2018**

For any region and zone, we can setup a cluster using *spark-ec2-branch-2* script:

export AWS\_ACCESS\_KEY\_ID=<aws access key id>

export AWS\_SECRET\_ACCESS\_KEY=<aws secret key>

./spark-ec2 --key-pair=<region-pem> --identity-file=<region-pem>.pem --region=<region> --zone=<zone> --vpc-id=<peered vpc> --subnet-id=<subnet for that vpc> --instance-type=<type> -s <number> launch <name>

Regions are: **us-east-1 (N.V), us-west-2 (Oregon), eu-west-1 (Ireland)**

Zones are: **<Region><a/b/c> (e.g. us-east-1a)**

VPCs are: **Default VPC for N.V., “oregon” for Oregon, “ireland” for Ireland (these VPCs are *peered1* to each other)**

In this way, we can set up individual regional clusters in any three regions using *spark-ec2-branch-2* script. Unfortunately, the script does not work for Ohio. But we can setup 3 clusters in 3 different zones in each region, so we can have 9 different clusters in 9 different zones in 3 geo-locations. I don’t know about you but that is quite geo-distributed for me!

Let’s assume we have setup three different clusters in three different regions:

|  |  |  |
| --- | --- | --- |
| **N.V. (us-east-1)** | **Oregon (us-west-2)** | **Ireland (eu-west-1)** |
| Master | Master | Master |
| Worker1 | Worker1 | Worker1 |
| Worker2 | Worker2 | Worker2 |

We can check if we have a working cluster by running the following code (which will be our standard testing code):

spark/bin/spark-submit \

--class org.apache.spark.examples.SparkPi \

--master <spark-master-ip-for-webui>\

--executor-memory <#> \

--conf spark.driver.maxResultSize=0 --conf spark.network.timeout=4000s\

spark/examples/jars/spark-examples\_2.11-2.0.0.jar

If we have a working cluster, we can assign an *Elastic IP*2 to each of the master nodes only. Before assigning we should stop all running clusters because the public-dns will change after this operation. As we are not assigning Elastic IPs to slave nodes, we have to edit *slaves* files in their respective cluster and put their respective private-ips. Later, we will change *slaves* files for all nodes for geo-spark. After restarting the instances, the public-ip and public-dns of master nodes remain unchanged and even if public-ips and public-dns of slave nodes change, it will not cause problem to start/stop the individual Spark cluster.

2https://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/vpc-peering.html

1https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html

3https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html

**Geo-Spark:**

After our work is done with these different clusters, we can turn all of them into a single geo-distributed spark cluster in simple steps which are as follows:

1. We first have to choose a single master, let this is be Oregon’s master. If you have not already, stop all running clusters (i.e. *stop-all.sh*). Copy everything from Oregon-master’s ssh folder and paste into every node in different cluster (masters included) ssh folder:

scp -i "<region-pem>.pem" .ssh/\* root@<public-dns-of-the-node>:~/.ssh/

Doing so will enable Oregon-master and the node, N (to what we are copying), to do *ssh* to each other. The communication will work both ways now because N now has *authorized\_keys* for the private key *id\_rsa* in Oregon-master and Oregon-master has the same *authorized\_keys* for the same private key *id\_rsa* in the other node! Also, in the *authorized\_keys*, we have the public key for the *pem* file of Oregon-master. So, if the N is from N.V., we can now *ssh* into that node from our terminal using *pem* file of Oregon-master. After completing this step, each and every node will have the same files, i.e. *authorized\_keys, id\_rsa, id\_rsa.pub, known\_hosts*.

1. Update the security group’s rules in all clusters so that it allows transmissions from each other. The easiest way to do it is to allow *All Traffic* in both *Inbound* and *Outbound* rules.

1. Replace /root/spark-ec2/cluster-url in /spark/conf/spark-env.sh.in Oregon’s master by the actual value (which can be found in that directory). Now copy *spark-env.sh* file to **all other nodes**. You can also collect private-ips of all nodes and name them *slave01, slave02…,* etc. in the master’s hosts file to make this process a little bit easier. You have to collect the private-ips of all other nodes anyway for updating *slaves* file, so rather doing it here may save some effort.
2. Collect private-ips of all nodes, and put them in /spark/conf/slaves.in Oregon’s master. Now copy this file to **all other nodes**.
3. Unfortunately, *start-all.sh* script does not work well for all nodes (especially the ones from different locations). A way around it is to start master by running:

spark/sbin/start-master.sh

and start each slave individually by:

spark/sbin/start-slave.sh spark://<public-dns-of-master>:7077

1. Run the test code to test if we have a working cluster. If you do, run experiments as you will. Stop the cluster by running a single call (*stop-all.sh*). You can stop all the instances. If you start the instances later, you can start the cluster by following the previous step. As we have elastic IP assigned to each master, we don’t have to change any files/codes. Also as each slave is now linked via private-ips (which also does not change), we should have no trouble to start the cluster. As each master has fixed public-ip now, we can also run our methods, TallnWide, easily by using same code.

**Multiple Spark Clusters using same nodes:**

1. For running multiple spark clusters, we have to note that each node is running a process of spark already, we have to start master and slaves in different process in different ports. For that lets copy everything from existing /spark folder to new folder, let’s name it /spark\_master. Do this for every node. Change /spark/conf/spark-env.sh. and /spark/conf/slaves. as necessary.
2. For sanity purpose, create a new terminal using *tmux.* There are four basic commands you need to remember
   1. Tmux to create a new terminal
   2. tmux attach -t <#> to go to an existing terminal
   3. ctrl+b d to get back to original terminal
   4. exit destroy tmux terminal you’re currently in

Use *tmux*  to better manage new spark programs and commands. Whether you use tmux or not, run

export SPARK\_IDENT\_STRING=<anything\_you\_want\_to\_name>

& for master

spark/sbin/start-master.sh --port <newport:M> --webui-port <another\_newport>

or for slave

spark/sbin/start-slave.sh spark://<public-dns-of-master>:M <newport> --webui-port <another\_newport>

And you will have another spark running in parallel using the same nodes (test if you have a working cluster)! One thing should be noted that using this, you have to manually stop processes (by replacing the word *start* by *stop* in above commands) to stop the cluster, because *stop-all.sh*  will not work properly and it will actually stop the other cluster.

**Geo-Hadoop:**

1. Steps of making a Geo-Hadoop are fairly simple too! Remember that *spark-ec2-branch-2*already has set up a nice Hadoop for us (*ephemeral-hdfs*). All the necessary configurations are done here.

The steps are similar to geo-spark. First stop all running Hadoop across clusters by running

ephemeral-hdfs/sbin/stop-dfs.sh && ephemeral-hdfs/sbin/stop-yarn.sh

1. Simply make one of the nodes a master, and put private-ips of all others in /ephemeral-hdfs/conf/slaves.
2. Delete the files in the directory from all other nodes /mnt/ephemeral-hdfs/data.

rm -r /mnt/ephemeral-hdfs/data/\*

1. Copy everything from the configuration folder to every other nodes:

scp -r ephemeral-hdfs/conf/\* root@<other\_nodes>:~/ephemeral-hdfs/conf/

1. Start all running Hadoop across clusters by running

ephemeral-hdfs/sbin/start-dfs.sh && ephemeral-hdfs/sbin/start-yarn.sh

1. For setting up spark with geo-hadoop, *spark-ec2-branch-2* has done the hard work for us, spark is already setup with this hdfs. The configuration file in *spark-env.sh* has been setup in such a way that nothing is hard-coded/ip-bound. So, it means any individual spark cluster can work with geo-hadoop. For testing using spark application, go to https://github.com/saagie/example-java-read-and-write-from-hdfs/blob/master/src/main/java/io/saagie/example/hdfs/Main.java and run it. Usually the name node uri is hdfs://<public\_dns\_of\_hadoop\_master>:9000, , so change the code from the link as necessary. But for *parallel spark* remove two lines after *PUBLIC-DNS* in *spark-env.sh.* (because of conflicting hadoop: not tested)

That’s basically it. I hope everything will work out smoothly and we will be able to get results very fast!

An Application of Geo-Spark and Hadoop:

1. spark/bin/spark-submit --class org.qcri.sparkpca.WriteW --master spark://ip-10-2-154-61.us-west-2.compute.internal:7077 --conf spark.driver.maxResultSize=0 --conf spark.network.timeout=4000s --driver-memory 887m --executor-memory 887m --driver-java-options "-Di=/input/ -Do=output/ -Drows=20000 -Dcols=128 -Dpcs=10 -DWi=W2 -Dhdfsuri=hdfs://ec2-54-218-104-83.us-west-2.compute.amazonaws.com:9000 -Dwritepath=/user/hdfs/W/" test.jar
2. spark/bin/spark-submit --class org.qcri.sparkpca.WriteW --master spark://ip-10-0-76-111.eu-west-1.compute.internal:7077 --conf spark.driver.maxResultSize=0 --conf spark.network.timeout=4000s --driver-memory 887m --executor-memory 887m --driver-java-options "-Di=/input/ -Do=output/ -Drows=20000 -Dcols=128 -Dpcs=10 -DWi=W1 -Dhdfsuri=hdfs://ec2-54-218-104-83.us-west-2.compute.amazonaws.com:9000 -Dwritepath=/user/hdfs/W/" test.jar
3. spark\_master/bin/spark-submit --class org.qcri.sparkpca.AccumW --master spark://ip-10-2-154-61.us-west-2.compute.internal:8001 --conf spark.driver.maxResultSize=0 --conf spark.network.timeout=4000s --driver-memory 887m --executor-memory 887m --driver-java-options "-Di=/input/ -Do=output/ -Drows=20000 -Dcols=128 -Dpcs=10 -DWi=W -Dhdfsuri=hdfs://ec2-54-218-104-83.us-west-2.compute.amazonaws.com:9000 -Dwritepath=/user/hdfs/W/" test.jar