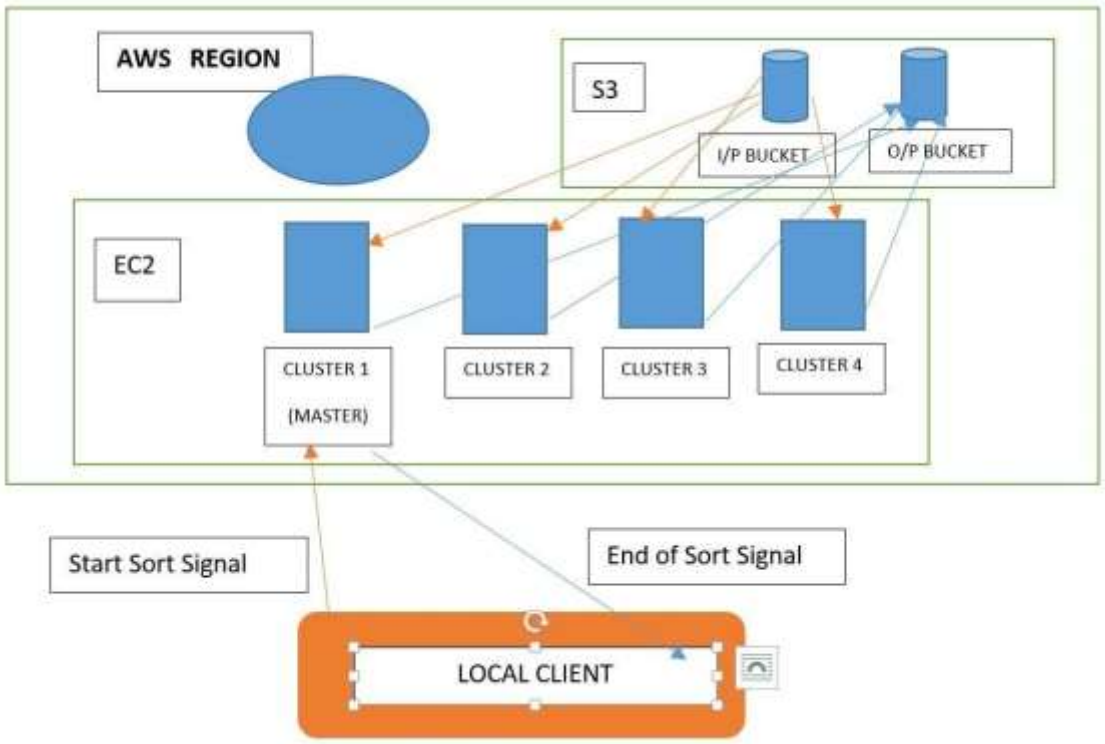
Assignment:-Distributed08 EC2 Sorting

Design Architecture:



Above diagram is the general representation for a cluster with number of EC2 instances as 4

This design follows a client server architecture. We have the following deliverables provided in the submission folder:

start-cluster, stop-cluster, splitFileList.sh, dataShipper.sh, startsort.sh, execute.sh, makessh.sh, server.sh, startExportToS3.sh, exportToS3.sh, sort, config

JAVA files - SortNode.java, TextSocket.java, WebClient.java, WebServer.java

MakeFile for aws setup for EC2 and ssh configuration

The entire architecture is based on a complex client-server socket programming with a single client and multiple EC2 servers (Instances) that can communicate with each other (bi-directional communication) for data and message passing for across.

The client waits for the entire procedure to be done and closes the connection once it receives the acknowledgement from the Master Server.

The Master Server does the task of connecting with the local client and other (N-1) servers (other instances on EC2 in our cluster, here N is the total number of EC2 instances)

The entire process is automated via bash scripting both interactive and background jobs that helps us establish this entire architecture.

The final output is received by the client as the top 10 values for the parameter – Dry Bulb temp

We can also see the output files generated on all the instances on EC2 server to the s3 bucket: (This bucket file is the input to the sort script)

A) **DESIGN DECISIONS:**

ARCHITECTURE:

1) Out of the total number of the clusters being populated, the first EC2 instance becomes Master node while the rest become Slave nodes.

2) **Role of Master**:

It gets the splitters of each slave node, processes these splitters and broadcast the new splitters to all slave nodes. The other role is to implement Sample Sort Algorithm, and sort the data received by the instance.

3) **Role of Slave:**

Send the splitters generated by the data available at its instance, receive new splitters broadcasted by the Master node. The other role is to implement Sample Sort Algorithm, and sort the data received by the instance.

4) **How data is received by the EC2 Instance**:

The local client gets the size of input bucket “s3://cs6240sp16/climate” which is 1.9 GB. Then it partitions the chunk into approximately equal size. Here size = (Total size of input file)/(number of cluster to be instantiated).

The client populates input files for the EC2 instances. This input file contains list of files that needs to be read by EC2 Node. The above logic of partitioning makes sure that each node reads approximately equal size of data. Each EC2 Node loads the data from S3 bucket to the EC2 node by reading this input file. Loading data from S3 bucket to EC2 node is done by FileChunkloader class.

5) **Sample Sort Algorithm:**

a) Consider p = number of EC2 instances.

b) Each instances reads (total size of data from s3 bucket / p). The client ensures that each EC2 instance reads approximately equal amount of data. For the given input of total size 1.9 GB of data, each EC2 instance reads approximately 1 GB of data for clusters = 2 and 250 MB of data for clusters = 8.

c) The program reads one file at a time and store each record as a Record object. The entire data for the EC2 instance is stored in the Arraylist of Record objects.

d) We sort the list of Records using Collections.Sort() , which internally uses quicksort. Sorting is done on “Dry Bulb Temp”

e) On the sorted list, we select (p-1) splitters (Dry Bulb Temp values) which are equidistant in the sorted list.

f) The splitters populated by the slaves are broadcasted to masters.

g) Master node receives splitter list from all the slaves. It merges this received splitters with its own splitters.

h) Then these splitters (populated in the last step) are again sorted and new (p-1) splitters are selected which are equidistant in the sorted splitter list.

i) The new splitters generated by step (e) are broadcasted to all slaves.

j) The master node and slave nodes insert these new splitters into their existing sorted Record list (generated in step - d) using binary search.

k) Then Master as well as Slaves, distribute these buckets to appropriate nodes. This is many to many Node communication which is handled by acknowledgement logic (Discussed in the challenges).

l) Once each EC2 instance finishes receiving its buckets from all the nodes except itself, the received buckets are again sorted and written to output[clusterId].txt (eg : output1.txt)

m) When all slaves complete step – I, they send acknowledgement signal to Master that they completed sorting process.

n) Once Master node receives Acknowledgement signal from all the slaves, the Master node sends “done” acknowledgement to the server running on master node.

o) After receiving the acknowledgement from all the slaves, the Master then communicates a ACK signal to the client which is then responsible for closing the TCP connection.

p) The final output i.e. top ten values of the required parameter “ dry bulb temp” is retrieved in the local file and the whole data file (sorted) is stored in the s3 bucket

B) **CHALLENGES:**

a) ***Binary Search***: The new splitters broadcasted by Master to slave nodes needed insertion into existing sorted list of Records. For this we used Binary Search and finding the appropriate index of the splitter in the given list was important.

b) ***Barrier Implementation***: Since this is a Master Slave architecture, we wanted to make Master keep listening to port until splitter list from all the slaves is received. For this we maintained a threshold count which made sure that data from all the slaves is received by master and then, further processing takes place.

c***) Acknowledgements***: Acknowledgement was one of the major challenge we faced to make data transfer synchronous between EC2 Nodes. So we implemented receiveAck () and sendAck() functions to send and receive Acknowledgements across the nodes. This was used in distributeBuckets() function. Suppose, Node 1 is receiving data from all the nodes, it sends acknowledgement to all nodes from which it receives data. After receiving acknowledgement, the remainder process of sending

bucket to other node is carried out. Acknowledgements helped us to regularize transfer of data between nodes and ensured that no data is loss during communication.

d) ***Multithreading***: There was a case where we wanted Master node to open certain number of ports simultaneously and start listening. To achieve this challenge, we implemented multithreading which ensured that all the ports are opened simultaneously and data is received.

e) ***Server/Client Implementation***: Since there was only one JAVA program "SortNode.java" running on each EC2 instance and the requirement was that the program should listen as well as send data through socket, we wanted to distinguish this feature. This was achieved by assigning cluster id to each node and cluster id =0 was made master. The logic of when to accept/receive data was manages accordingly.