Big Data Administration

# Installing Hadoop

We first installed Hadoop in Pseudo-Distributed Mode, setting up the required six core Hadoop daemons (listed below) on a single virtual machine before expanding out to others.

* NameNode
* SecondaryNameNode
* DataNode
* ResourceManager
* NodeManager
* JobHistoryServer

We began by choosing a theme for our cluster. Each group of virtual machines was named after either cats, birds, dogs or reindeer and assigned unique IP addresses. These settings were altered in the vagrant file and eth-1 & eth-2 configuration files. Once these had been configured, we installed vagrant on our computers to create four virtual machines each. We then exported these machines and re-imported them into VirtualBox.

To set things in motion, we created a repository file to point the hosts towards the Cloudera CDH5 installation files. Once Hadoop had been installed, we began setting up its core HDFS (Hadoop Distributed File System) daemons. We started the NameNode, SecondaryNameNode and DataNode on our primary machines and created a HDFS staging directory and a YARN (Yet Another Resource Negotiator) log directory. Following this, we started the YARN and MapReduce daemons. To confirm which services were running, we used the command:

sudo jps

Now that everything we needed was up and running, we tested setup by uploading a text file to a new directory in HDFS with specific file permissions. This was successful; our Hadoop installation had worked!

# Creating a Small Hadoop Cluster

For our next exercise, we configured a small Hadoop cluster on our individual machines.

We stop all running services and removed unnecessary daemons and log files. We then reinstalled them on different hosts according to the layout we had been given.

In order for our cluster to function properly, each virtual machine needed to access the four Hadoop configuration files:

core-site.xml | hdfs-site.xml | yarn-site.xml | mapred-site.xml

Amongst other things, we added a number of properties to these files which pointed towards the host locations for different daemons. We also created a new file named Hadoop-env.sh which would reduce the heap size of Hadoop daemons; this is necessary as our virtual machines had limited RAM. We then copied the newly created or altered files to the other virtual machines in our cluster.

In the next stage, we began setting up the Hadoop file system, creating a number of different directories with the appropriate ownership for YARN, HDFS, MapReduce and the user.

Whilst checking logs, we started up the HDFS, YARN and MapReduce daemons according to the given structure. After everything had been started, we tested the setup once more with a simple word count MapReduce job. This was again successful. We tracked the status of the job through the ResourceManager Web UI.

# Creating a Largeish Hadoop Cluster

We then began creating a plan for our cluster, taking account of the resources and functionality required by each node. We designated eight machines to run both a DataNode and a NodeManager.

The DataNodes are where the majority of data processing takes place; they are responsible for storing, reading and writing data. The NodeManagers have a number of functions; they manage and monitor container processes (launching ApplicationMasters and processes), communicate with the ResourceManager (sending resource information and container status, along with regular heartbeats), provide logs to HDFS and maintain node-level security.

The NameNode stores and manages the namespace and metadata (fsimage), and monitors the slave nodes. In order to avoid having a Single Point of Failure, we chose to implement a High Availability system. This We installed a standby NameNode that would run on a separate rack and takeover from the primary NameNode in the event of failure. In this setup, the Active NameNode writes the fsimage and edits to a quorum of JournalNodes while the Standby NameNode reads from the JournalNodes and periodically performs checkpointing, where it retrieves the fsimage and edits from the NameNode, loads the metadata, applies changes, creates a new fsimage and returns this to the NameNode.

To support concurrency and provide additional failure protection, we also set up Zookeeper on our cluster.

ZooKeeper provides an infrastructure for cross-node synchronisation and can be used by applications to ensure that tasks across the cluster are synchronised. This is achieved by maintaining status type information in memory on ZooKeeper servers, which keep a copy of system state and persist this information in local log files.

Since we were aiming to simulate a large Hadoop cluster, we chose to support our system with multiple ZooKeeper servers, including a master server. In this setup, each client machine communicates with one of the ZooKeeper servers to retrieve and update its synchronisation information. In this way, distributed processes are able to coordinate with each other through a shared hierarchical name space of data registers, known as znodes. This architecture allows ZooKeeper to provide high throughput and availability with low latency, but the size of the database that ZooKeeper can manage is limited by memory.

Since the Zookeeper Failover Controllers are lightweight and work closely with the NameNodes, we set these up on the same hosts with an additional controller on a separate machine to make an odd number. In addition, we configured a Zookeeper quorum on five hosts.

Moving on, we decided to use MapReduce Version 2, which works with a ResourceManager and JobHistoryServer. We installed both applications on different machines to the two NameNodes to \_\_\_\_\_\_\_\_\_\_\_.

The ResourceManager runs a scheduler to determine resource allocation, tracks heartbeats from other nodes, manages ApplicationMasters and allocates - and de-allocates - containers according to memory, CPU and data locality. The JobHistoryServer, on the other hand, is responsible for servicing all job history related requests from clients. Accessing its Web UI gives detailed information and runtime metrics for individual jobs.

In our plan, we designated four CentOS machines as client hosts. On these, we set up HiveServer2, Impala shells and GreenPlum clients.

Apache Hive is a high-level abstraction on MapReduce. It uses HiveQl for querying; however, it is not as efficient as Impala.

Impala is a high-performance SQL engine which queries data in HDFS using HiveQL and Hive Metastore. On each of the Ubuntu machines, we installed an Impala server for fast data processing. Installing Impala on the same machines running DataNodes brings the processing to the data, limiting the amount of time and resources needed to complete jobs.

Impala also requires a single Catalog and State Store per cluster. Since both are lightweight, we installed these on one of the Ubuntu machines running DataNodes.

## Problems & Solutions

Naturally, we ran into a number of issues while setting up and configuring our Hadoop cluster. Most of these were communication issues related to firewalls and IP addresses and could easily be solved with the following commands:

sudo service network restart

sudo service iptables stop / sudo ufw disable

It is important to note, however, that these are only workarounds and not solutions. Ideally, we would configure the firewalls so that the machines in our cluster were able to communicate but none outside could. Unfortunately, we were unable to implement this solution in the given timeframe.

# Extract, Transform & Load Processing

This section of the course focused on extracting, transforming and loading data. We initially worked on the MovieLens data set, which we downloaded to a group directory in HDFS.

We began by setting up Pig to work within our cluster, installing the Grunt shell on our client hosts. This interpreter converts PigLatin scripts into MapReduce jobs to submit to the cluster. The main advantage of using Pig to execute jobs is the simplicity with which you can request complex jobs such as joins.

We experimented with loading the data into Pig and reading it, understanding how Pig interpreted data without a pre-defined schema. We then moved on to more complex processes, such as grouping and filtering the data, taking samples and calculating new variables, finally carrying out a complex join.

We were then set a challenge of working with two large JSON of over one million entries, containing Amazon product information and reviews. We were tasked with loading, cleaning and sorting this data.

The first problem we ran into was a lack of space on our virtual machines. To solve this, we had to extend the virtual Disk images for each Ubuntu virtual machine since they are where the data is stored. In order to do this, we carried out the following steps for each Ubuntu machine:

1. Shut down the virtual machine
2. Clone the VDMK file to VDI
3. Resize the VDI file to allow for 200GB of storage (as opposed to 20GB)
4. Clone the VDI file back to VDMK
5. Swap the old disk to the new disk in Virtual box
6. Boot the virtual machine from a LiveCD image
7. Use GParted Partition Editor to configure partitions
8. Power off the virtual machine
9. Remove LiveCD from the virtual drive

The second issue we came across was the speed at which our queries were processed, no matter the sample size; most jobs were progressing at such a slow rate that they would take around five hours to complete. In order to tackle this issue, we looked again at the \_\_\_\_\_\_\_\_\_ files. We noticed that each virtual machine had been allocated more memory than was available (\_\_\_\_\_) so we adjusted these values to \_\_\_\_\_\_\_\_\_. This decreased the Pig processing time substantially, and we were able to run most queries within ten minutes. However, the same was still much longer when outputting the entire data set.

http://www-01.ibm.com/software/data/infosphere/hadoop/zookeeper/