# APACHE HADOOP ON OPENSTACK CLOUD

Project Report submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology

in

**Information Technology**

under the Supervision of

***PROF. DR. SATYA PRAKASH GHRERA***

By

***HIMANSHI SHARMA - 111451***

to

Jaypee University of Information Technology,

Waknaghat, Solan – 173234, Himachal Pradesh

May, 2015

# Certificate

This is to certify that project report entitled "Apache Hadoop on OpenStack Cloud (Hadoop as a Service)", submitted by Himanshi Sharma in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science & Engineering to Jaypee University of Information Technology, Waknaghat, Distt. Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

**PROJECT SUPERVISOR**

**Date:**

**Prof. Dr. Satya Prakash Ghrera**

**Professor, Brig (Retd.)**

**Head, Dept. of CSE**

# ACKNOWLEDGEMENT

The satisfaction that accompanies the successful completion of any task would be incomplete without the mention of people whose ceaseless cooperation made the training possible and whose constant guidance and encouragement crowned my efforts with success. None of this was possible without the support and hard work of the larger Apache Hadoop community and The OpenStack Foundation. I want to encourage all readers to get involved in the community and open source in general.

I am grateful to my PROJECT MENTOR PROF. DR. SATYA PRAKASH GHRERA and PROJECT COORDINATOR DR. HEMRAJ SAINI for their guidance, inspiration and constructive suggestions that helped me in the preparation of this project. They always helped me by providing with notes, software and other resources. They were my guiding light during the whole journey of the project and I can always look up to them. I also want to extend my gratitude to faculties of my department who helped in successfully completing the project.

Date: Himanshi Sharma

# TABLE OF CONTENTS

# LIST OF FIGURES

# ABSTRACT

Big Data at its core is simply a way of describing problems which are not solvable using traditional tools. Big Data is defined by its three Vs i.e. Volume, Velocity and Variety. This Big Data coming from wide variety of source can help make transnational decisions, and to be able to make these decisions on data of any scale it is important to access the right kind of tools. Apache Hadoop is such a framework designed to store, process and manage such a form of data accross cluster of computers. It is 100% open source, and pioneered a fundamentally new way of storing and processing data. Instead of relying on expensive, proprietary hardware and different systems to store and process data, Hadoop enables distributed parallel processing of huge amounts of data across inexpensive, industry-standard servers that both store and process the data, and can scale without limits. With Hadoop, no data is too big. At its core, Apache Hadoop is a framework for storing data on large clusters of commodity hardware— everyday computer hardware that is affordable and easily available— and running applications against that data. A cluster is a group of interconnected computers (known as nodes) that can work together on the same problem. Using networks of affordable compute resources to acquire business insight is the key value proposition of Hadoop. But of course, the business model is generally spilt across various dimensions, and each dimension may have their own Hadoop Clusrer, which means deploying different storage for each of the different cluster of which some data can be same in different cluster also if there is a need of merging of two dimensions for some analysis it is required to load the combined data into a new cluster which is again a time consuming and expensive task. This is where OpenStack Swift container comes into picture, Swift provides a Object storage which allows to store files or objects. Swift architectures is build in such a way that can be used directly with Hadoop clusters, which allows it provides a 'centralized' storage, which can be accessed by any cluster directly and any cluster can use it to store the data, also projects like pig, hive that are used with Hadoop can also access the Swift storage directly so as to process the data, hence removing the need of storing redundant data and help encouraging innovations because of its low cost and low complexity. Both Apache Hadoop and OpenStack Swift represent open source projects, Swift servers as an excellent storage use-case for Hadoop, which can be remarkably used to process Big Data in a more efficient scalable way.

**KEYWORDS**:
Big Data, Apache Hadoop, Cluster, Nodes, Jobs, Trackers, OpenStack Swift Container,Objects.

# WHY DO WE NEED THIS ?

The most commonly used storage systems in the enterprise data center today are filesystem storage and block level storage. Filesystem storage is typically deployed as Network Attached Storage (NAS) systems and used for storing and sharing files over a network. Block storage is typically deployed as Storage Area Network (SAN) systems and appears to an operating system like locally attached drives, which is required for running things like databases. For the applications your developers are writing today, this storage model has several drawbacks which, according to the industry analyst firm Gartner, includes:

- It doesn’t efficiently scale to support new workloads
- It's bogged down by operational overhead
- It's difficult to match storage to application requirements
- It's time-consuming to adjust to workload changes and migrations
- It's manually managed, or at best semi-automated

These drawbacks become increasingly important as the rapid growth of unstructured data that many, if not most, enterprises are experiencing today continues. Nearly every industry is storing and serving more data at higher fidelity to an increasing number applications and users.

Hence today’s view of the ‘future’ of computing revolves around data storage and elastic, distributed platforms. Cloud computing has good manageability and resource utilization. This makes tend to move services to clouds: use public or build private ones. There are some platforms for building private clouds: Cloudstack, Openstack, Eucalyptus, VMware. Despite private clouds benefit there are problems with compatibility with existing systems. This imposes restrictions is use. The main idea was to make compatibility with Openstack Swift and most popular distributed computation framework - Hadoop.

## *Hadoop*

Hadoop supports main storages such as Amazon S3, Kosmos, Hadoop Archive Filesystem. Hadoop can run MapReduce programs over that storages, integration is fully compatible. But there are variety of other storages that Hadoop doesn't support.

Developers can write a driver to support custom FS, but some features, for instance Hadoop data locality, will be missing. This causes network, CPU or memory overhead.

***Swift***

Swift is cloud storage developing by Openstack community. Swift is Amazon S3 analogue: both systems have accounts, accounts contain buckets (containers in Swift), buckets (containers) contain objects. Objects are real files of any kind. For data processing in S3 Amazon offers Elastic MapReduce service, Swift doesn't have such compatibility, and it servers as an excellent use-case for centralizing the data storage and reduces the network, CPU or memory overheads. Hence, merging two technology gives an edge answering the needs of 'future' computing.

# 1. INTODUCTION & OBJECTIVES OF PROJECT

## 1.1 Introduction to System

- ✓ This Hadoop tool will help anyone to setup Hadoop cluster and use it.
- ✓ It uses an open source Hadoop framework which allows user to tailor it as per its needs and requirements.
- ✓ Tool has one interface to control and manage the entire network of nodes and perform all the functions.

## 1.2 Scope of System

**Cost:**

The cost required in making the tool is not very high. It includes costs of one system having quad core processor, 8GB RAM and Linux operating system.

**Effort:**

Most of the effort is involved in coding and designing part which included coding the servers and cluster and MapReduce coding and attaching the HDFS to Swift container of OpenStack Cloud.

**Time:**

The required in making this tool was around 4 months and most of the time will be consumed in testing the tool based on the bandwidth of Local Area Network.

## 1.3 Objectives

The main objective of the tool is to automate the process of cluster setup and management. We aim to create a tool that will ask for client preferences and then will setup up Hadoop Distributed File System consisting of namenode and datanodes, which will use Swift Container of OpenStack Cloud for storage purpose. The tool will also setup Map Reduce framework consisting of jobtrackers and tasktrackers, that will again use space from Swift Container. The client can store data into Hadoop Cluster using the tool itself and can also carry out Hadoop jobs using the tool.

# 2. Tools / Platform / Language

Selection of right tools obviously improves the process of system development & quality of results. The central benefit of using proper tools is the ability to organize system information. Tools provide an efficient environment for creation, storage, manipulation, management & documentation of system.

## 2.1 Hardware Requirements

**Client:**

- ✓ RAM :- 1 GB
- ✓ PROCESSOR :- At least two 2 GHz and above processors
- ✓ STORAGE :- 3 GB

**Cluster:**

- ✓ Systems:- min 4 systems
- ✓ RAM:- 4GB on each system
- ✓ STORAGE:- 1TB on each system

## 2.2 Software Requirements

**Operating system:**

RedHat Enterprise Linux Server Edition or Ubuntu system

**Platform:**

- ✓ *Front end* : Graphical User Interface using shell scripting or Web Based user interface using Python CGI.
- ✓ *Back end* : Python scripting, SQL.

# 3. Design and Implementation

Apache Hadoop is an industry standard and widely adopted MapReduce implementation. As aim of the project is to enable users to easily provision and manage Hadoop clusters on OpenStack. It is worth mentioning that Amazon provides Hadoop for several years as Amazon Elastic MapReduce ( EMR) service, which serves as the inspiration of this project.

Hadoop helps in storing data over a distributed network where data files of huge size is divided into large number of blocks and these bocks are stored on swift container. Hadoop MapReduce is used to process these blocks of data in parallel. Python scripting is language which has been used to make this tool along with HTML & CSS and makes use of this graphical user interface to interact with user. The workflow and design of the tool goes like this :

***Scanning of the network:***

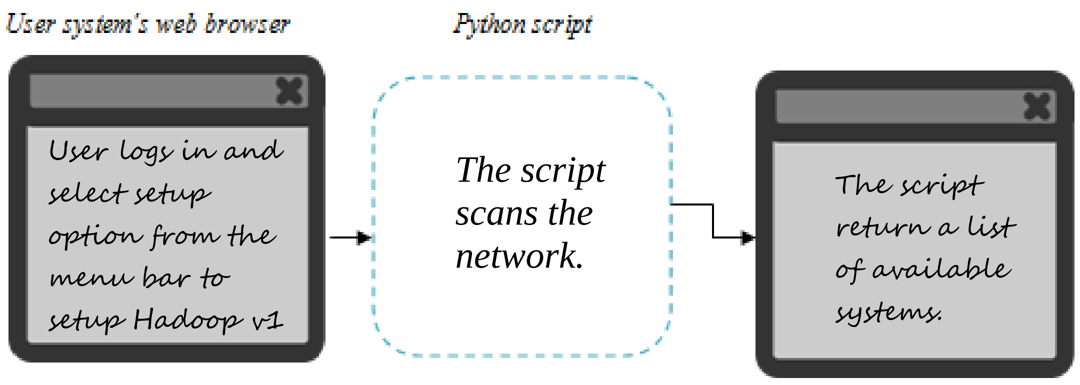

User system's web browser
Python script
User logs in and select setup option from the menu bar to setup Hadoop v1
The script scans the network.
The script return a list of available systems.

*Figure - 1 Scanning of the network*

The user selects the Setup option from the drop down menu form the menu bar, script then scans the entire network to check for the available alive system in the network that can be either established as a namenode or a datanode. After completely scanning the entire network the script returns a list ip address of the available system in the network on which the namenode or datanode can be established, this list of ip addresss is saved as a text file and also rendered on the browser's screen of the user's system.

***Checking system for memory, RAM and core processor**:*

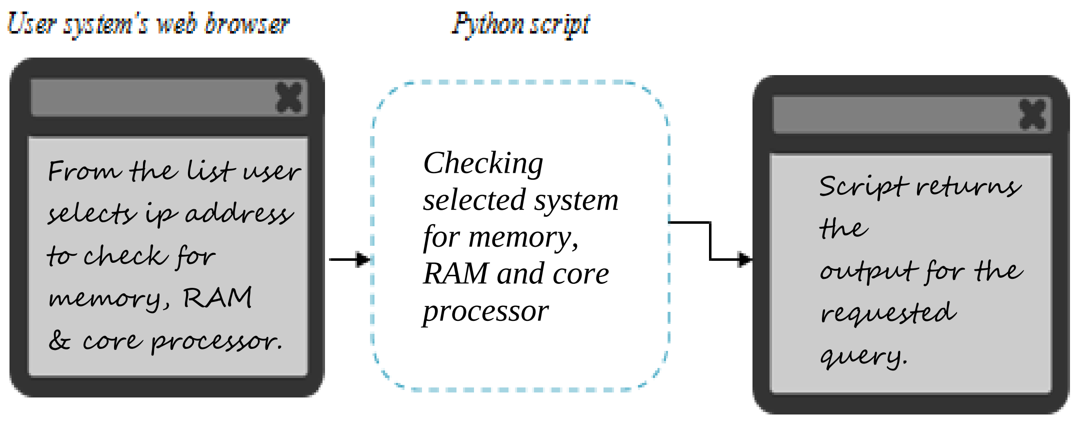

User system's web browser
Python script
From the list user selects ip address to check for memory, RAM & core processor.
Checking selected system for memory, RAM and core processor
Script returns the output for the requested query.

*Figure - 2 Checking selected system for its system detail*

After rendering the list of ip on the screen, the user selects some ip from the list to get a full detail of the system. For each ip selected, the script login to the system via secure shell (ssh) to get detail about hard-disk, memory and core-processor of the system and present it to the user via rendering it on the bowser's screen of the user's system. This information of the system selected by user, helps the user to decide on which system he/she should establish a node as either namenode or datanode on, this basically gives the overview of the selected system which user can use to select best possible system for establishing namenode or datanode from the network as per his/her requirements.

*Establishing node as namenode:*

The information about memory, RAM & cpu cores of selected system that is rendered on the screen is first used to establish a node as namenode.

The suitable system from the network is selected that user wants to act as namenode in his/her's Hadoop cluster. The user usually select one node from the network to act as namenode in the Hadoop cluster, as the cluster requires only one namenode for maintaining the metadata but user can select more than one if they want. Moreover, a node can be used to act as a substitute namenode which holds the same data as that of original namenode, if in case the original namenode gets corrupted by any reasons the data will not be lost completely, as then the substitute node will replace the original corrupted namenode, this basically ensures the integrity of the data.

Selected nodes ip addresses are saved in a namenode list, which is used by script to install packages and configure Hadoop cluster. This list will contain the list of ip addresses of the namenode in the Hadoop cluster. Following are the steps to configure a node from the network to act as namenode.

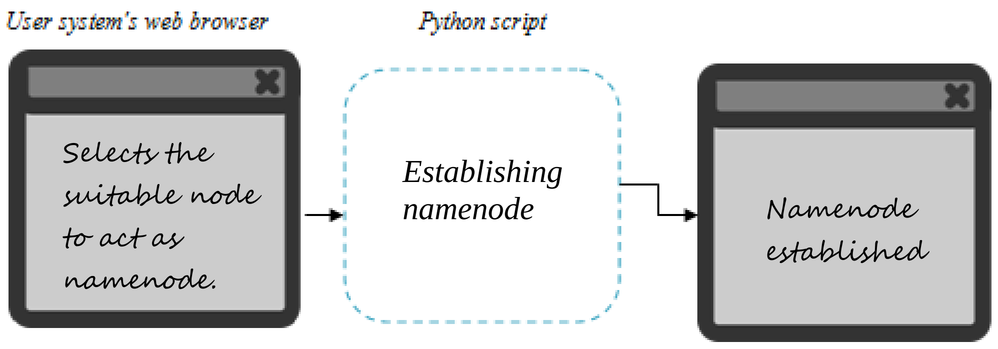

User system's web browser
Python script
Selects the suitable node to act as namenode.
Establishing namenode
Namenode established

*Figure - 3 Establishing node as namenode.*

*i) Installing necessary packages on the selected system*:

The first step to establish a node as namenode is to install Hadoop and jdk ( Java Development Toolkit version 1.7.0_51) packages on that node. A Hadoop package is required as it is essential element in deploying the Hadoop cluster, it contains necessary modules and jdk required as it provides necessary environment for deploying Hadoop cluster, as Hadoop is written in java and it requires jdk7 and later version of jdk6. The namenode list that contains the ip addresses of the namenode are

used by the python script in accessing the system via ssh and installing the packages on the listed systems.
The storage requirements of the Hadoop Cluster can be fullfilled by either assigning a local partition of Hard Disk of the HDFS nodes (namenode and datanode) to the Cluster or by using the Swift Object storage of the OpenStack.

*ii)*

*Partioning for allocating space:*

Hadoop cluster have a distributive file system and the file system require space in order to store and process Big data. The storage location can be either a part of the local system where the user is establishing the node or we can use Swift container as storage location for HDFS and for running the jobs.

Similarly, a node can be established as datanode through same series of steps using python script, user can select one or more nodes to act as a datanode in the cluster as per his/her requirements. Like a namenode list, a datanode list is created by the scrip that contains the ip addresses of all the datanodes, which is further used by script in installing packages, allocating storage and configuring clusters.

*Swift for allocating space:*

To use OpenStack Swift Containter it is important that the OpenStack Swift is either installed on the namenode or on all the HDFS nodes (namenode or datanode) or it can be implemented on different set of network and used in HDFS through a web API.
A number of ways can be used to install the OpenStack swift, the one used to support the project is via packstack:
1. Installing openstack kernel for the system which will be used as the host for the installation. Appropriate package can be downloaded into the system and installed via yum.

```
[root@serverX ~]# yum install kernel*openstack*
```

2. Installing *openstack-packstack*, which is a utility tool to quickly deploy Red Hat OpenStack either interactively or non-interactively by creating and using an answer

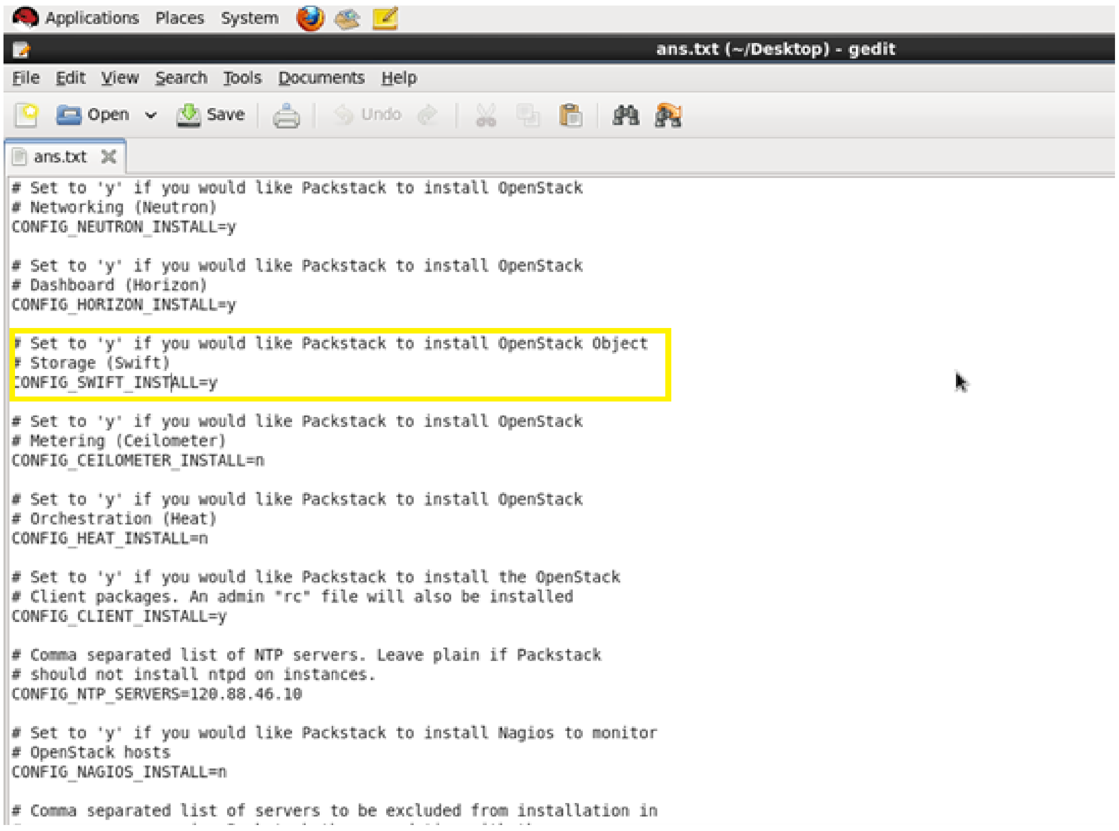
file that can be tuned. (Before installing it is important to check the kernel is openstack kernel, this is done by using command uname -r).

```
[root@serverX ~]# yum install -y openstack-packstack
```

3. After installing the packstack utility tool, generate the answer file and modify it to install only the necessary modules. For the project, Swift holds a prime importance, the parameters relating to the Swift can tuned via answer file.

```
[root@serverX ~]# packstack --gen-answer-file=a.txt
```

This *a.txt* file is used for installing OpenStack on to a system or systems via puppet module.



```
# Set to 'y' if you would like Packstack to install OpenStack
# Networking (Neutron)
CONFIG_NEUTRON_INSTALL=y

# Set to 'y' if you would like Packstack to install OpenStack
# Dashboard (Horizon)
CONFIG_HORIZON_INSTALL=y

# Set to 'y' if you would like Packstack to install OpenStack Object
# Storage (Swift)
CONFIG_SWIFT_INSTALL=y

# Set to 'y' if you would like Packstack to install OpenStack
# Metering (Ceilometer)
CONFIG_CEILOMETER_INSTALL=n

# Set to 'y' if you would like Packstack to install OpenStack
# Orchestration (Heat)
CONFIG_HEAT_INSTALL=n

# Set to 'y' if you would like Packstack to install the OpenStack
# Client packages. An admin "rc" file will also be installed
CONFIG_CLIENT_INSTALL=y

# Comma separated list of NTP servers. Leave plain if Packstack
# should not install ntpd on instances.
CONFIG_NTP_SERVERS=120.88.46.10

# Set to 'y' if you would like Packstack to install Nagios to monitor
# OpenStack hosts
CONFIG_NAGIOS_INSTALL=n

# Comma separated list of servers to be excluded from installation in
```

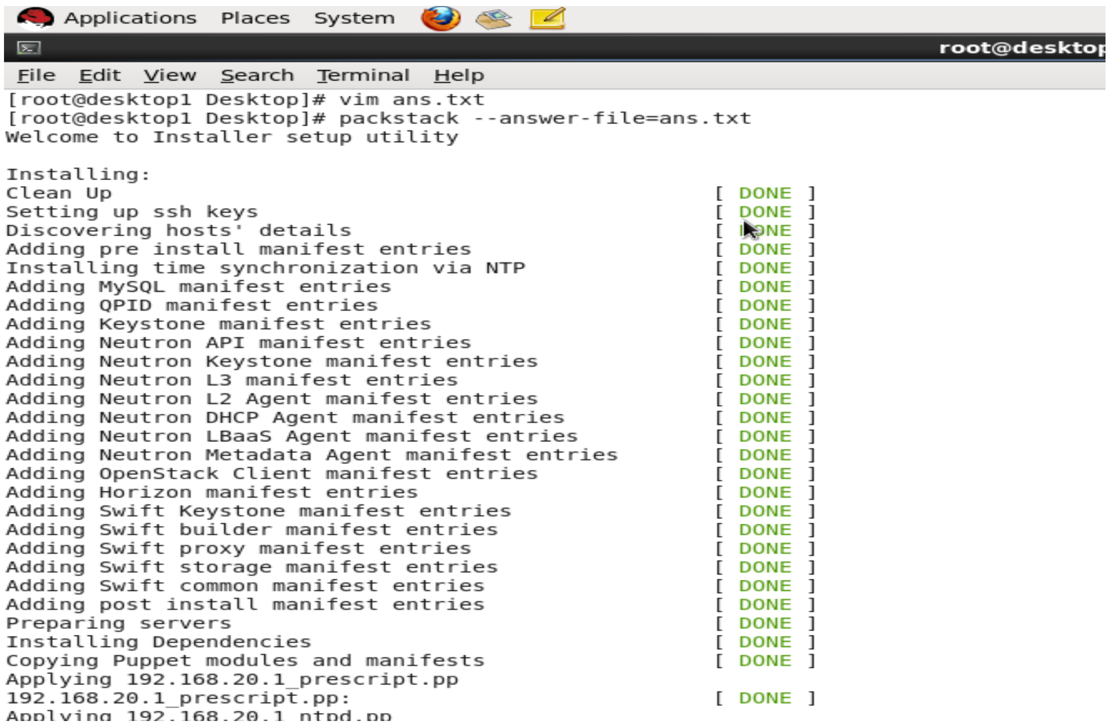
*Figure - 4 Answer file generated by packstack to install OpenStack packages.*

The user can install different packages as per the requirement by just mentioning 'y' in front of the specific variable.

For the project Swift prameters like *config_proxy_server*, *config_storage_server*, config*_replication*, etc can be tuned by providing, the ip addresses of the system on which the proxy server and storage server will be deployed against the specific parameter in the answer file.

4. After tuning the all the necessary parameters, the answer file is used to install the various packages on the system whose ip addresses are included in the answer file.

```
[root@serverX ~]# packstack --answer-file=a.txt
```


Applications Places System
root@desktop
File Edit View Search Terminal Help

```
[root@desktop1 Desktop]# vim ans.txt
[root@desktop1 Desktop]# packstack --answer-file=ans.txt
Welcome to Installer setup utility

Installing:
Clean Up                                             [ DONE ]
Setting up ssh keys                                  [ DONE ]
Discovering hosts' details                           [ DONE ]
Adding pre install manifest entries                  [ DONE ]
Installing time synchronization via NTP              [ DONE ]
Adding MySQL manifest entries                        [ DONE ]
Adding QPID manifest entries                         [ DONE ]
Adding Keystone manifest entries                     [ DONE ]
Adding Neutron API manifest entries                  [ DONE ]
Adding Neutron Keystone manifest entries             [ DONE ]
Adding Neutron L3 manifest entries                   [ DONE ]
Adding Neutron L2 Agent manifest entries             [ DONE ]
Adding Neutron DHCP Agent manifest entries           [ DONE ]
Adding Neutron LBaaS Agent manifest entries          [ DONE ]
Adding Neutron Metadata Agent manifest entries       [ DONE ]
Adding OpenStack Client manifest entries             [ DONE ]
Adding Horizon manifest entries                      [ DONE ]
Adding Swift Keystone manifest entries               [ DONE ]
Adding Swift builder manifest entries                [ DONE ]
Adding Swift proxy manifest entries                  [ DONE ]
Adding Swift storage manifest entries                [ DONE ]
Adding Swift common manifest entries                 [ DONE ]
Adding post install manifest entries                 [ DONE ]
Preparing servers                                    [ DONE ]
Installing Dependencies                              [ DONE ]
Copying Puppet modules and manifests                 [ DONE ]
Applying 192.168.20.1_prescript.pp
192.168.20.1_prescript.pp:                           [ DONE ]
Applying 192.168.20.1_ntpd.pp
```

*Figure - 5 Installing OpenStack packages via puppet module.*

The installation will take few minutes, after which Swift can be accessed via a Web API (through a dashboard), or via command line python swift-client.

To use this with Hadoop, following code is added to *core-site.xml :*

```
<property>
<name>fs.swift.impl</name>
<value>org.apache.hadoop.fs.swift.snative.SwiftNativeFileSystem</valu
e>
</property>
```

***Configuring the cluster:***

After completing the task of establishing nodes as namenode and datanode on the network and for the Hadoop cluster, the script configures the cluster so as to make each node aware of the namenode of the cluster, this is done by configuring the core-site.xml configuration file on every node on the network that are established as namenode and datanode.

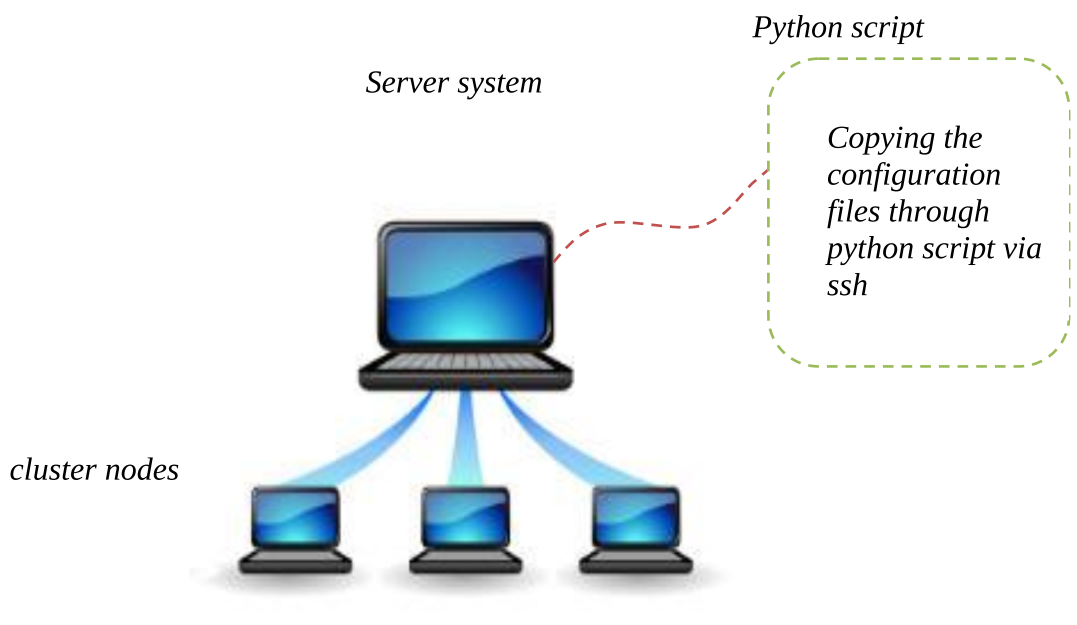
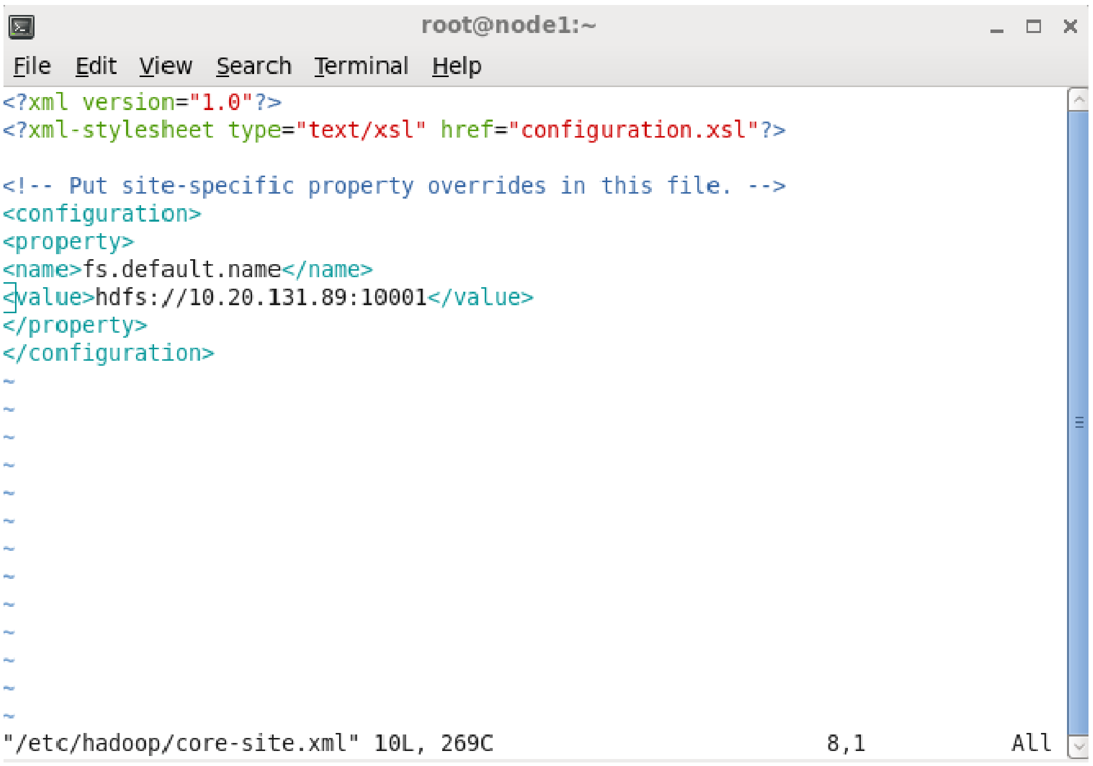

Python script
Server system
Copying the configuration files through python script via ssh
cluster nodes

*Figure - 6 Copying configuration files from server system to nodes to configure the Hadoop cluster*

The script locally generates the configuration files using the namenode list and then copy it to the other nodes, as the core-site.xml file will contain the same content i.e. it defines the namenode of the cluster which is same for every node.



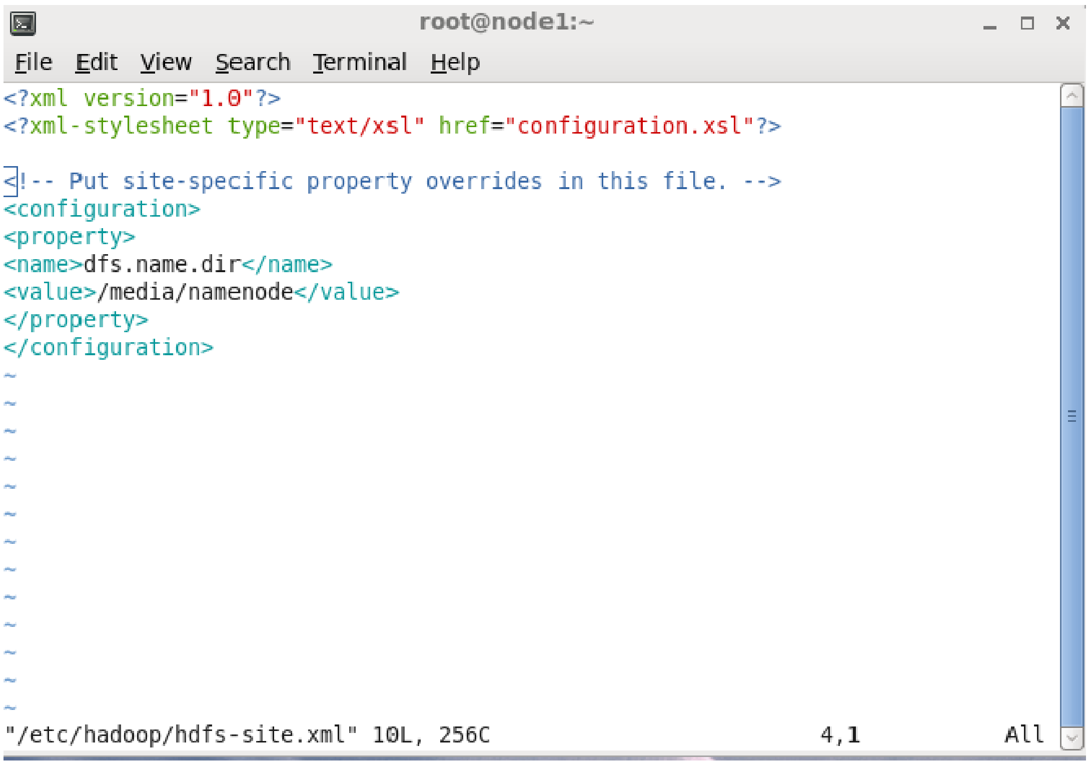
```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
<name>fs.default.name</name>
<value>hdfs://10.20.131.89:10001</value>
</property>
</configuration>
~
~
~
~
~
~
~
~
~
~
~
~
~
"/etc/hadoop/core-site.xml" 10L, 269C                    8,1           All
```

*Figure - 7 Configuration file : core-site.xml*

This configuration file *'core-site.xml'* defines the file system's default namenode, which other datanodes will communicate in order to get instructions and to send heartbeats ( *signal the namenode that the respected datanode who sent the signal is alive* ). The script copy this file to each and every node via *secure copy* (scp) after generating it locally.

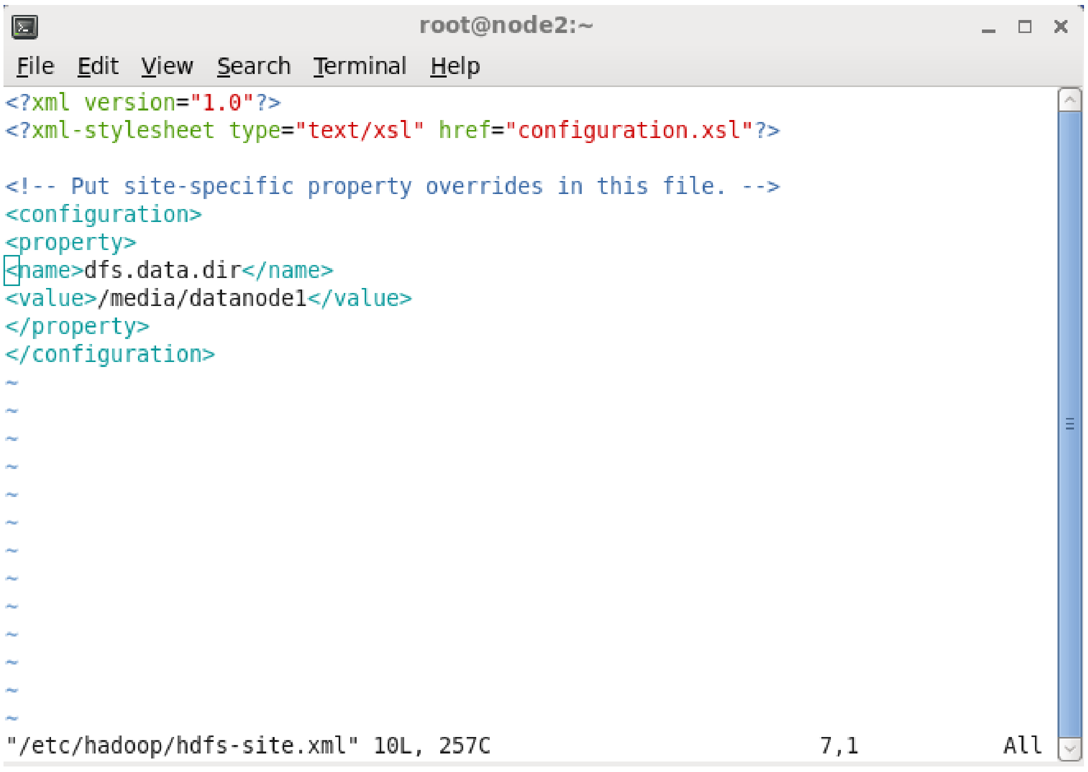
Other important file for configurating cluster is *hdfs-site.xml* which defines the storage location for every node and it is different for every node as every node advertise different storage location which is either local to it or it uses a *swift* container.
The scripts generated this file locally as well for each and every node and copy it via *secure copy* (scp) to every node.



```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
<name>dfs.name.dir</name>
<value>/media/namenode</value>
</property>
</configuration>
~
~
~
~
~
~
~
~
~
~
~
~
~
"/etc/hadoop/hdfs-site.xml" 10L, 256C                        4,1           All
```

*Figure -  8 Configuration file : hdfs-site.xml, for local storage location on namenode.*

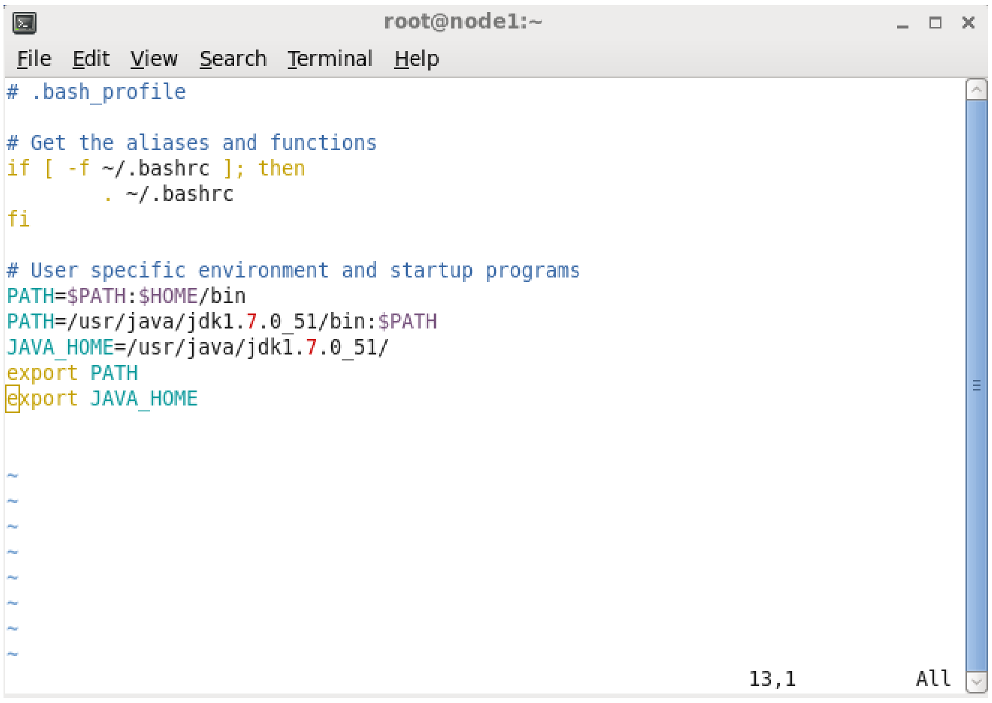


```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
<name>dfs.data.dir</name>
<value>/media/datanode1</value>
</property>
</configuration>
~
~
~
~
~
~
~
~
~
~
~
~
~
"/etc/hadoop/hdfs-site.xml" 10L, 257C                        7,1           All
```

*Figure -  9 Configuration file : hdfs-site.xml, for local storage location on datanode.*

These file defines the distributive file system's name and data nodes directory or storage location. Apart from these two file one more file needs to be edited in every node is *.bashrc* or *.bash_profile*, which defines the java home and path for specific user environments necessary to run daemons on the node.



```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs
PATH=$PATH:$HOME/bin
PATH=/usr/java/jdk1.7.0_51/bin:$PATH
JAVA_HOME=/usr/java/jdk1.7.0_51/
export PATH
export JAVA_HOME



~
~
~
~
~
~
~
~
                                                     13,1          All
```

*Figure - 10 .bash_profile : define the path and java home.*

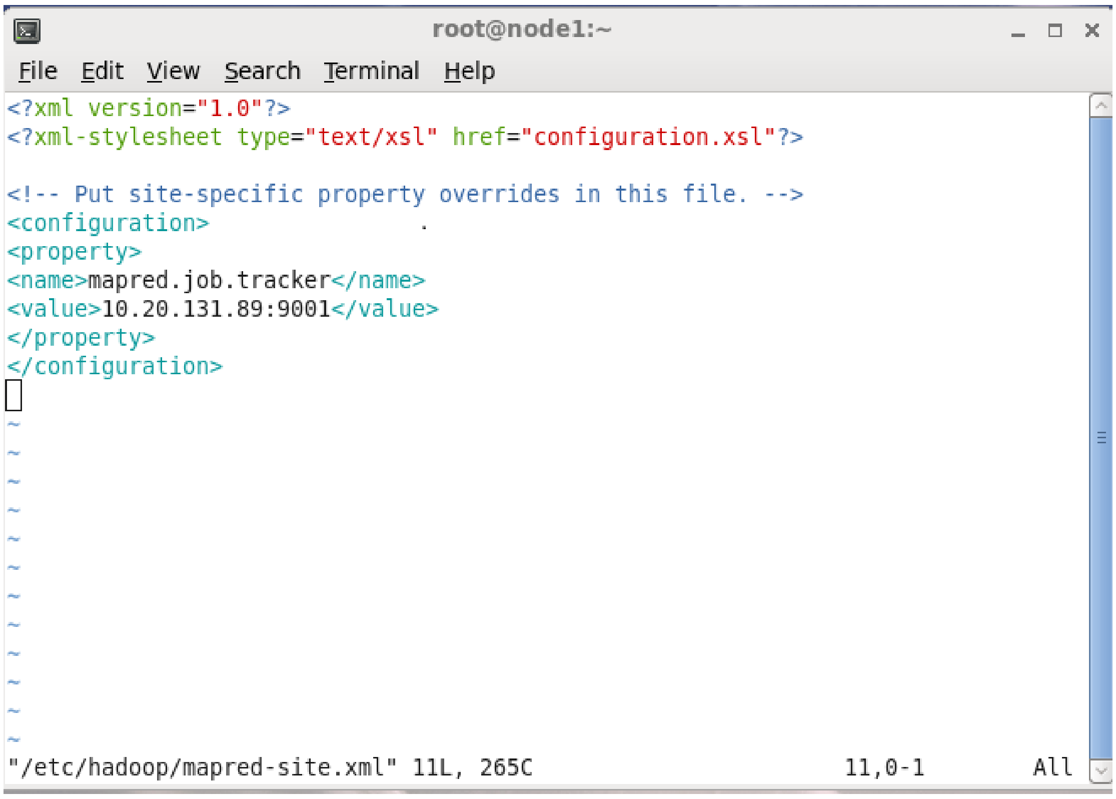
This file is again generated locally by the script and the script copies it to other nodes via *secure copy* (scp)

After completing these tasks the cluster is completely configured to store huge files or the Big data through namenode.

***Establishing and configuring MapReduce Framework:***

In a similar fashion the Job Tracker and Task Tracker are established in the cluster for processing purposes, the script first scans the network returns the available nodes that user can establish as Job Tracker and Task Tracker, it then checks the selected nodes and returns the list of nodes with their RAM, Hard disk, CPU processors, the user makes a selection from the list.

After this the configuration file i.e. *mapred-site.xml* is transferred to each of the node for the configuration of Job Tracker and Task Tracker.



```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
<name>mapred.job.tracker</name>
<value>10.20.131.89:9001</value>
</property>
</configuration>

~
~
~
~
~
~
~
~
~
~
~
~
"/etc/hadoop/mapred-site.xml" 11L, 265C                        11,0-1        All
```

*Figure - 11 Configuration file : mapred-site.xml, to define the job tracker.*

After the configuration process, the script can be used to start task trackers and job trackers, and can be used to run job on the data present on the local disk of the nodes or in the OpenStack Swift Container.

Projects like Hive and Pig can be used with Hadoop to run the job and to process the variety and volumes of data.

# 4. Detailed Description of Technology used

## 4.1 Apache Hadoop

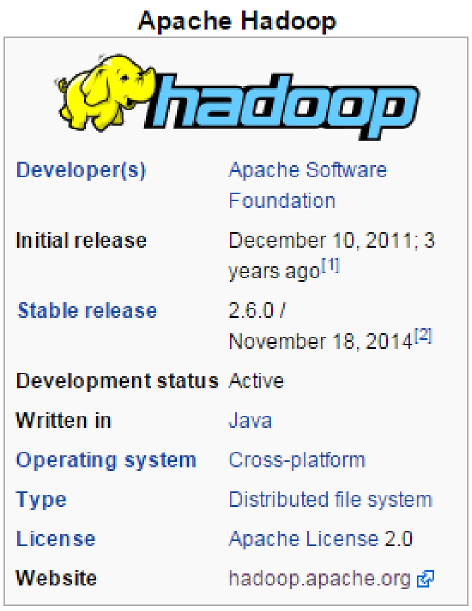
Apache Hadoop is a registered trademark of the Apache Software Foundation.

The Apache™ Hadoop® project develops open-source software for reliable, scalable, distributed computing.

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

The project includes these modules:

- **Hadoop Common**: The common utilities that support the other Hadoop modules.
- **Hadoop Distributed File System (HDFS™)**: A distributed file system that provides high-throughput access to application data.
- **Hadoop YARN**: A framework for job scheduling and cluster resource management.
- **Hadoop MapReduce**: A YARN-based system for parallel processing of large data sets.



Apache Hadoop

| | |
|---|---|
| Developer(s) | Apache Software Foundation |
| Initial release | December 10, 2011; 3 years ago[1] |
| Stable release | 2.6.0 / November 18, 2014[2] |
| Development status | Active |
| Written in | Java |
| Operating system | Cross-platform |
| Type | Distributed file system |
| License | Apache License 2.0 |
| Website | hadoop.apache.org |

*Figure - 12*

Out of these components two of them are main components, which are: a distributed processing framework named *MapReduce* (which is now supported by a component called *YARN (Yet Another Resource Negotiator)*, which we describe a little later) and a distributed file system known as the *Hadoop Distributed File System*, or *HDFS*. An application that is running on Hadoop gets its work divided among the nodes (machines) in the cluster, and HDFS stores the data that will be processed. A Hadoop cluster can span thousands of machines, where HDFS stores data, and MapReduce jobs do their processing near the data, which keeps I/O costs low.

YARN stands for "Yet Another Resource Negotiator" and was added later as part of Hadoop 2.0. YARN takes the resource management capabilities that were in MapReduce and packages them so they can be used by new engines. This also streamlines MapReduce to do what it does best, process data. With YARN, you can now run multiple applications in Hadoop, all sharing a common resource management. As of September, 2014, YARN manages only CPU (number of cores) and memory, but management of other resources such as disk, network and GPU is planned for the future.

For the end-users, though MapReduce Java code is common, any programming language can be used with "Hadoop Streaming" to implement the "map" and "reduce" parts of the user's program. *The Hadoop framework itself is mostly written in the Java programming language, with some native code in C and command line utilities written as shell-scripts.*

MapReduce is extremely flexible, and enables the development of a wide variety of applications. Apache Pig, Apache Hive, Apache Spark among other related projects expose higher level user interfaces like Pig Latin and a SQL variant respectively, which can be used in accordance with MapReduce and HDFS in order to get useful information from structured Big Data by running a job through MapReduce over this data which is stored in HDFS. The result of a job run in Hadoop may be used by an external system, may require further processing in a legacy system, or the processing requirements might not fit the MapReduce paradigm. Any one of these situations will require data to be exported from HDFS. One of the simplest ways to download data from HDFS is to use the Hadoop shell.

Sqoop is an Apache project that is part of the broader Hadoop ecosphere. In many ways Sqoop is similar to distcp (*Moving data efficiently between clusters using Distributed Copy recipe*). Both are built on top of MapReduce and take advantage of its parallelism and fault tolerance. Instead of moving data between clusters, Sqoop was designed to move data from and into relational databases using a JDBC driver to connect. Hadoop is a platform that provides both distributed storage and computational capabilities. Hadoop was first conceived to fix a scalability issue that existed in Nutch, an open source crawler and search engine. At the time Google had published papers that described its novel distributed file system, the Google File System (GFS), and Map-Reduce, a computational framework for parallel processing. The successful implementation of these papers' concepts in Nutch resulted in its split into two separate projects, the second of which became Hadoop, a first-class Apache project.

Looking at Hadoop from an architectural perspective, as shown in Figure 1.2, it is a distributed master-slave architectures, the master node consists of the namenode, secondary namenode, and jobtracker daemons (the so-called *master daemons*). In addition, this is the node from which you manage the cluster for the purposes of this demonstration (using the Hadoop utility and browser). The slave nodes consist of the tasktracker and the datanode (the slave daemons). The distinction of this setup is that the master node contains those daemons that provide management and coordination of the Hadoop cluster, where the slave node contains the daemons that implement the storage functions for the Hadoop file system (HDFS) and MapReduce functionality (the data processing function).

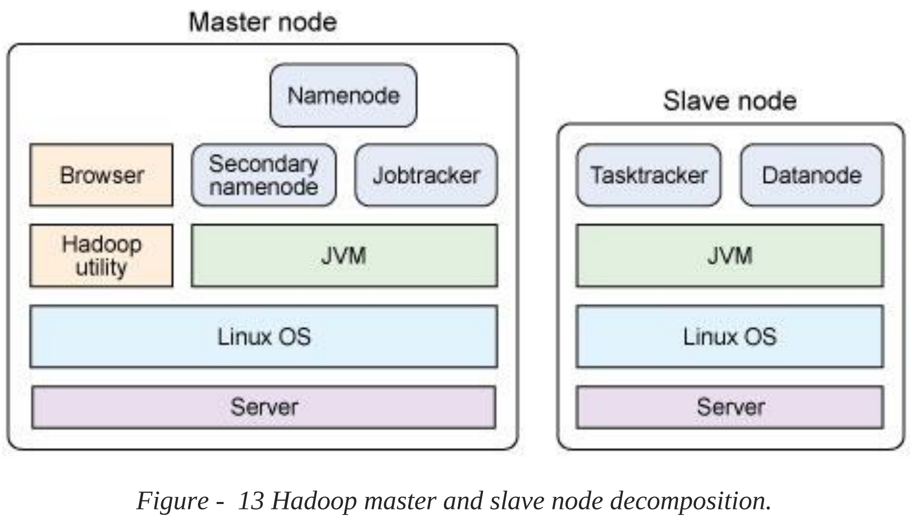


*Figure - 13 Hadoop master and slave node decomposition.*

Looking at Hadoop Version 2 ecosystem , as shown in figure 14, it that consists of the Hadoop Distributed File System (HDFS) for storage and Map-Reduce for computational capabilities that forms the base of the Hadoop architecture.

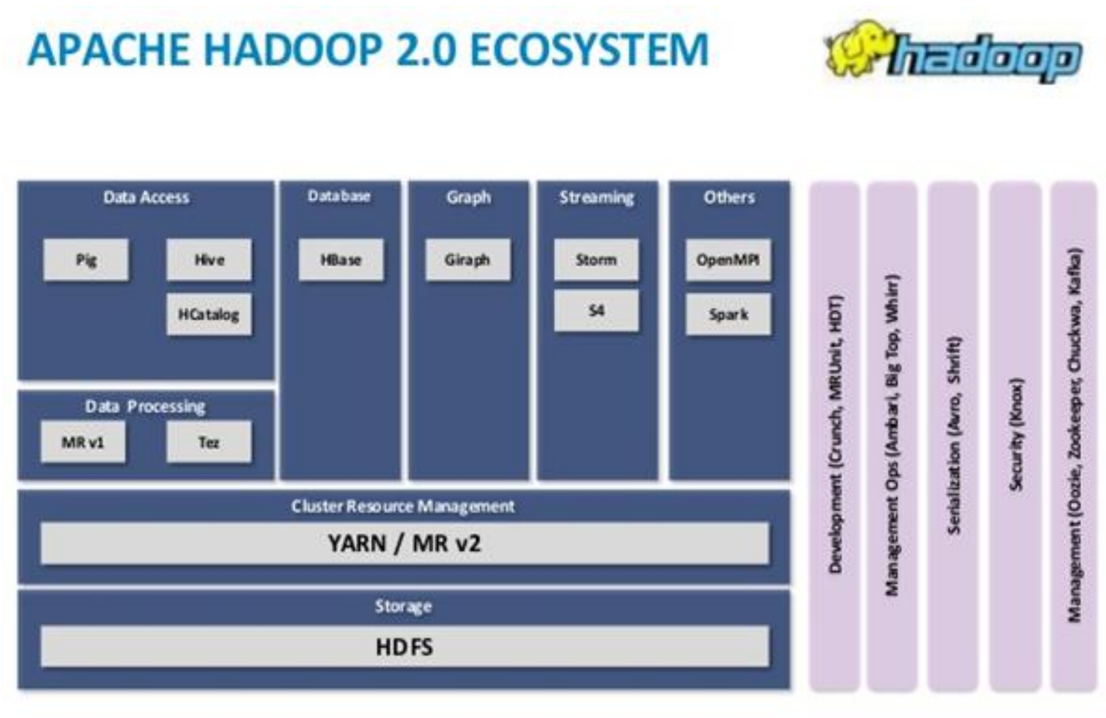

APACHE HADOOP 2.0 ECOSYSTEM
hadoop
Data Access
Pig
Hive
HCatalog
Database
HBase
Graph
Giraph
Streaming
Storm
S4
Others
OpenMPI
Spark
Data Processing
MR v1
Tez
Cluster Resource Management
YARN / MR v2
Storage
HDFS
Development (Crunch, MRUnit, HDT)
Management Ops (Ambari, Big Top, Whirr)
Serialization (Avro, Shrift)
Security (Knox)
Management (Oozie, Zookeeper, Chuckwa, Kafka)

*Figure - 14 Apache Hadoop 2.0 Ecosystem.*

Traits intrinsic to Hadoop are data partitioning and parallel computation of large datasets. Its storage and computational capabilities scale with the addition of hosts to a Hadoop cluster, and can reach volume sizes in the petabytes on clusters with thousands of hosts.

## 4.2 HDFS

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets in other words HDFS was built to support high throughput, streaming reads and writes of extremely large files..

Traditional large storage area networks (SANs) and network attached storage (NAS) offer centralized, low-latency access to either a block device or a filesystem on the order of terabytes in size. These systems are fantastic as the backing store for relational databases, content delivery systems, and similar types of data storage needs because they can support full-featured POSIX semantics, scale to meet the size requirements of these systems, and offer low-latency access to data.

Imagine for a second, though, hundreds or thousands of machines all waking up at the same time and pulling hundreds of terabytes of data from a centralized storage system at once. This is where traditional storage doesn't necessarily scale. By creating a system composed of independent machines, each with its own I/O subsystem, disks, RAM, network interfaces, and CPUs, and relaxing (and sometimes removing) some of the POSIX requirements, it is possible to build a system optimized, in both performance and cost, for the specific type of workload we're interested in.

There are a number of specific goals for HDFS:

• Store millions of large files, each greater than tens of gigabytes, and filesystem sizes reaching tens of petabytes.

• Use a scale-out model based on inexpensive commodity servers with internal JBOD ("Just a bunch of disks") rather than RAID to achieve large-scale storage. Accomplish availability and high throughput through application-level replication of data.

• Optimize for large, streaming reads and writes rather than low-latency access to many small files. Batch performance is more important than interactive response times.

• Gracefully deal with component failures of machines and disks.

• Support the functionality and scale requirements of MapReduce processing.

While it is true that HDFS can be used independently of MapReduce to store large datasets, it truly shines when they’re used together. MapReduce, for instance, takes advantage of how the data in HDFS is split on ingestion into blocks and pushes computation to the machine where blocks can be read locally.

**Design**

HDFS, in many ways, follows traditional filesystem design. Files are stored as opaque blocks and metadata exists that keeps track of the filename to block mapping, directory tree structure, permissions, and so forth. This is similar to common Linux filesystems such as ext3. So what makes HDFS different?

Traditional filesystems are implemented as kernel modules (in Linux, at least) and together with userland tools, can be mounted and made available to end users. HDFS is what’s called a userspace filesystem. This is a fancy way of saying that the filesystem code runs outside the kernel as OS processes and by extension, is not registered with or exposed via the Linux VFS layer. While this is much simpler, more flexible, and arguably safer to implement, it means that you don't mount HDFS as you would ext3, for instance, and that it requires applications to be explicitly built for it. In addition to being a userspace filesystem, HDFS is a distributed filesystem. Distributed filesystems are used to overcome the limits of what an individual disk or machine is capable of supporting. Each machine in a cluster stores a subset of the data that makes up the complete filesystem with the idea being that, as we need to store more block data, we simply add more machines, each with multiple disks. Filesystem metadata is stored on a centralized server, acting as a directory of block data and providing a global picture of the filesystem’s state. Another major difference between HDFS and other filesystems is its block size. It is common that general purpose filesystems use a 4 KB or 8 KB block size for data. Hadoop, on the other hand, uses the significantly larger block size of 64 MB by default. In fact, cluster administrators usually raise this to 128 MB, 256 MB, or even as high as 1 GB. Increasing the block size means data will be written in larger contiguous chunks on disk, which in turn means data can be written and read in larger sequential operations. This minimizes drive seek operations—one of the slowest operations a mechanical disk can perform—and results in better performance when doing large streaming I/O operations.

Rather than rely on specialized storage subsystem data protection, HDFS replicates each block to multiple machines in the cluster. By default, each block in a file is replicated three times. Because files in HDFS are write once, once a replica is written, it is not possible for it to change. This obviates the need for complex reasoning about the consistency between replicas and as a result, applications can read any of the available replicas when accessing a file. Having multiple replicas means multiple machine failures are easily tolerated, but there are also more opportunities to read data from a machine closest to an application on the network. HDFS actively tracks and manages the number of available replicas of a block as well. Should the number of copies of a block drop below the configured replication factor, the filesystem automatically makes a new copy from one of the remaining replicas. Throughout this book, we'll frequently use the term replica to mean a copy of an HDFS block.

Applications, of course, don't want to worry about blocks, metadata, disks, sectors, and other low-level details. Instead, developers want to perform I/O operations using higher level abstractions such as files and streams. HDFS presents the filesystem to developers as a high-level, POSIX-like API with familiar operations and concepts.

*HDFS Contains:*

- HDFS has master/slave architecture. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients.
- In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on.
- HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes.
- The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes.

- The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode. [1,2]

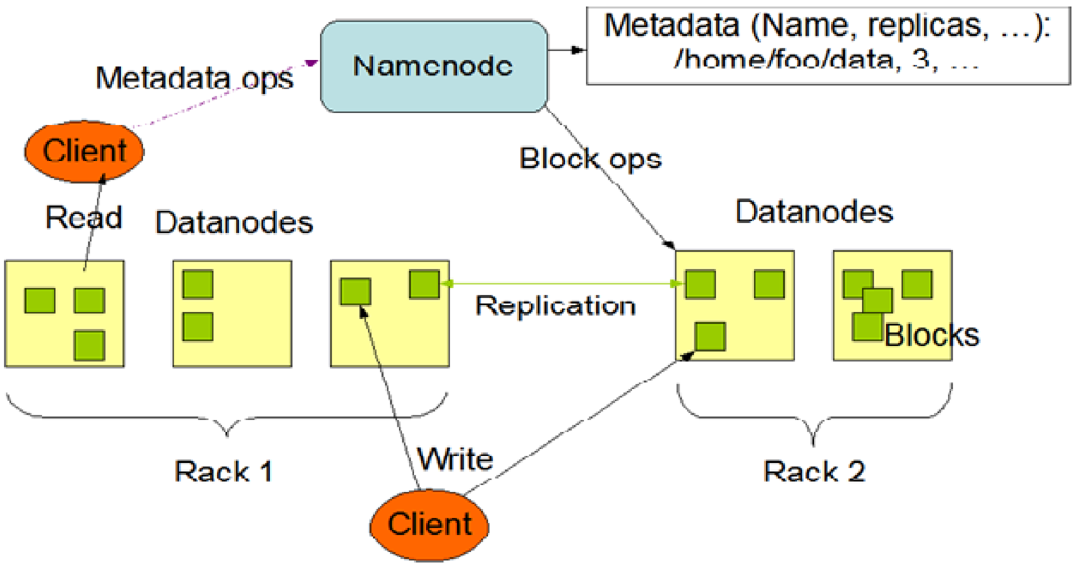


*Figure - 15 HDFS Architecture.*

## 4.3 MAPREDUCE

MapReduce refers to two distinct things: the programming model (covered here) and the specific implementation of the framework. Designed to simplify the development of large-scale, distributed, fault-tolerant data processing applications, MapReduce is foremost a way of writing applications. In MapReduce, developers write *jobs* that consist primarily of a *map function* and a *reduce function*, and the framework handles the gory details of parallelizing the work, scheduling parts of the job on worker machines, monitoring for and recovering from failures, and so forth. Developers are shielded from having to implement complex and repetitious code and instead, focus on algorithms and business logic. User-provided code is invoked by the framework rather than the other way around. This is much like Java application servers that invoke servlets upon receiving an HTTP request; the container is responsible for setup and teardown as well as providing a runtime environment for user-supplied code. Similarly, as servlet authors need not implement the low-level details of socket I/O, event handling loops, and complex thread coordination, MapReduce developers program to a well-defined, simple interface and the “container” does the heavy lifting.
The idea of MapReduce was defined in a paper written by two Google engineers in 2004, titled "MapReduce: Simplified Data Processing on Large Clusters" (J. Dean, S. Ghemawat). The paper describes both the programming model and (parts of) Google’s specific implementation of the framework. Hadoop MapReduce is an open source implementation of the model described in this paper and tracks the implementation closely. Specifically developed to deal with large-scale workloads, MapReduce provides the following features:

***Simplicity of development***

MapReduce is dead simple for developers: no socket programming, no threading or fancy synchronization logic, no management of retries, no special techniques to deal with enormous amounts of data. Developers use functional programming concepts to build data processing applications that operate on one record at a time. Map functions operate on these records and produce intermediate key-value pairs. The reduce function then operates on the intermediate key-value pairs, processing all values that have the same key together and outputting the result. These primitives can be used to implement filtering, projection, grouping, aggregation, and other common data processing functions.

***Scale***

Since tasks do not communicate with one another explicitly and do not share state, they can execute in parallel and on separate machines. Additional machines can be added to the cluster and applications immediately take advantage of the additional hardware with no change at all. MapReduce is designed to be a *share nothing* system.

***Automatic parallelization and distribution of work***

Developers focus on the map and reduce functions that process individual records (where “record” is an abstract concept—it could be a line of a file or a row from a relational database) in a dataset. The storage of the dataset is not prescribed by MapReduce, although it is extremely common, as we’ll see later, that files on a distributed filesystem are an excellent pairing. The framework is responsible for splitting a MapReduce job into tasks. Tasks are then executed on *worker nodes* or (less pleasantly) *slaves*.

***Fault tolerance***

Failure is not an exception; it’s the norm. MapReduce treats failure as a first-class citizen and supports reexecution of failed tasks on healthy worker nodes in the cluster. Should a worker node fail, all tasks are assumed to be lost, in which case they are simply rescheduled elsewhere. The unit of work is always the task, and it either completes successfully or it fails completely. In MapReduce, users write a *client application* that submits one or more *jobs* that contain user-supplied map and reduce code and a job configuration file to a cluster of machines. The job contains a *map* function and a *reduce* function, along with *job configuration* information that controls various aspects of its execution. The framework handles breaking the job into *tasks*, scheduling tasks to run on machines, monitoring each task’s health, and performing any necessary retries of failed tasks. A job processes an input dataset specified by the user and usually outputs one as well. Commonly, the input and output datasets are one or more files on a distributed filesystem. This is one of the ways in which Hadoop MapReduce and HDFS work together, but we'll get into that later.

***The Stages of MapReduce***

A MapReduce job is made up of four distinct stages, executed in order: client job submission, map task execution, shuffle and sort, and reduce task execution.

- **"Map" step:** Each worker node applies the "map()" function to the local data, and writes the output to a temporary storage. A master node orchestrates that for redundant copies of input data, only one is processed.
- **"Shuffle" step:** Worker nodes redistribute data based on the output keys (produced by the map ( ) function), such that all data belonging to one key is located on the same worker node.
- **"Reduce" step:** Worker nodes now process each group of output data, per key, in parallel.

Client applications can really be any type of application the developer desires, from commandline tools to services. The MapReduce framework provides a set of APIs for submitting jobs and interacting with the cluster. The job itself is made up of code written by a developer against the MapReduce APIs and the configuration which specifies things such as the input and output datasets. As described earlier, the client application submits a job to the cluster using the framework APIs. A master process, called the jobtracker in Hadoop MapReduce, is responsible for accepting these submissions (more on the role of the jobtracker later). Job submission occurs over the network, so clients may be running on one of the cluster nodes or not; it doesn't matter. The framework gets to decide how to split the input dataset into chunks, or input splits, of data that can be processed in parallel. In Hadoop MapReduce, the component that does this is called an input format, and Hadoop comes with a small library of them for common file formats. The following steps implements the stages of executing a job.

1. **Prepare the Map() input** – the "MapReduce system" designates Map processors, assigns the input key value *K1* that each processor would work on, and provides that processor with all the input data associated with that key value.

2. **Run the user-provided Map() code** – Map() is run exactly once for each *K1* key value, generating output organized by key values *K2*.

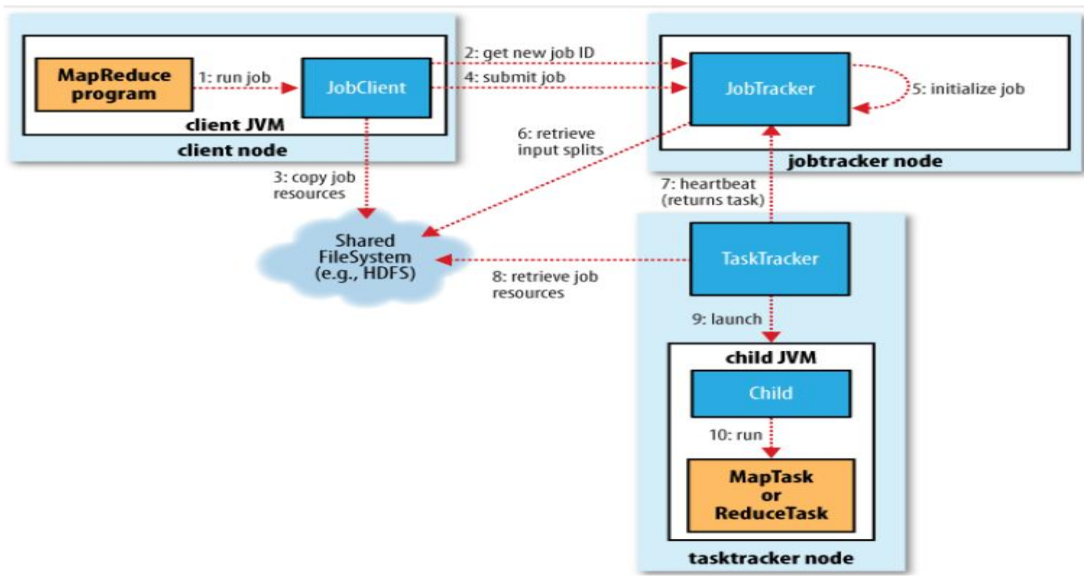


*Figure - 16 Classic Framework for job execution.*

3. **"Shuffle" the Map output to the Reduce processors** – the MapReduce system designates Reduce processors, assigns the *K2* key value each processor should work on, and provides that processor with all the Map-generated data associated with that key value.
4. **Run the user-provided Reduce() code** – Reduce() is run exactly once for each *K2* key value produced by the Map step.
5. **Produce the final output** – the MapReduce system collects all the Reduce output, and sorts it by *K2* to produce the final outcome [1,2]

## 4.4 OpenStack

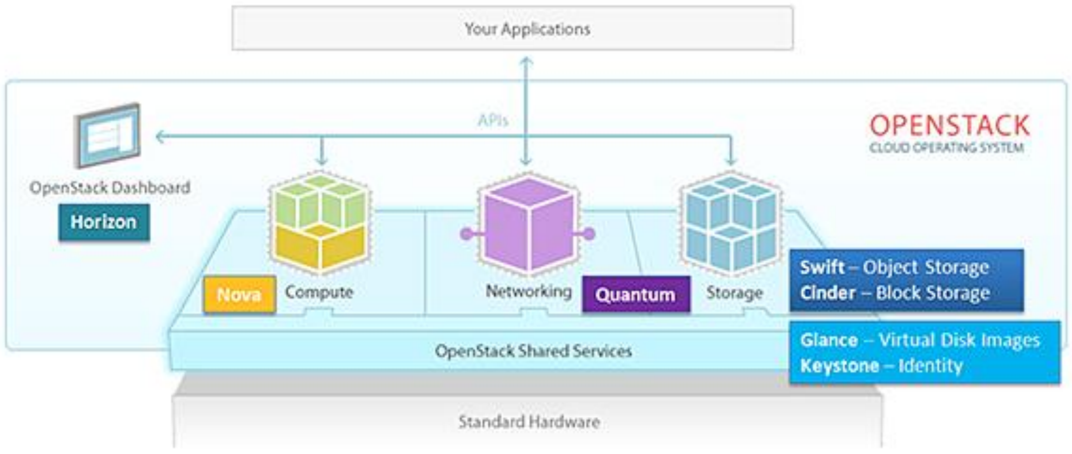
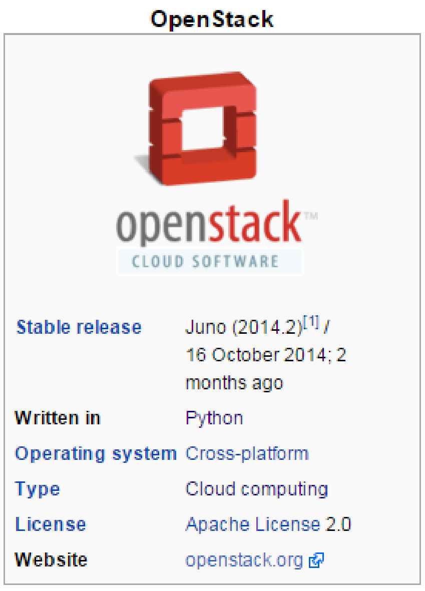


*Figure - 17 Core components of OpenStack architecture.*

OpenStack began in 2010 as a joint project of Rackspace Hosting and NASA. Currently, it is managed by the OpenStack Foundation, a non-profit corporate entity established in September 2012 to promote OpenStack software and its community. More than 200 companies have joined the project, including Arista Networks, Avaya, Canonical, Cisco, Dell, EMC, Ericsson, GoDaddy, Huawei, IBM, Intel, Mella nox, Mirantis and many more.



**OpenStack**

| | |
|---|---|
| **Stable release** | Juno (2014.2)[1] / 16 October 2014; 2 months ago |
| **Written in** | Python |
| **Operating system** | Cross-platform |
| **Type** | Cloud computing |
| **License** | Apache License 2.0 |
| **Website** | openstack.org |

*Figure - 18*

The long-term vision for OpenStack is to produce a ubiquitous open source cloud computing platform that meets the needs of public and private cloud providers regardless of size. OpenStack services control large pools of compute, storage, and networking resources throughout a data center.

The technology behind OpenStack consists of a series of interrelated projects delivering various components for a cloud infrastructure solution that control pools of processing, storage, and networking resources throughout a data center. Each service provides an open API so that all of these resources can be managed through a dashboard that gives administrators control while empowering users to provision resources through a web interface, a command-line client, or software development kits that support the API. Many OpenStack APIs are extensible, meaning you can keep compatibility with a core set of calls while providing access to more resources and innovating through API extensions. The OpenStack project is a global collaboration of developers and cloud computing technologists. The project produces an open standard cloud computing platform for both public and private clouds. By focusing on ease of implementation, massive scalability, a variety of rich features, and tremendous extensibility, the project aims to deliver a practical and reliable cloud solution for all types of organizations.

***Architecture:***

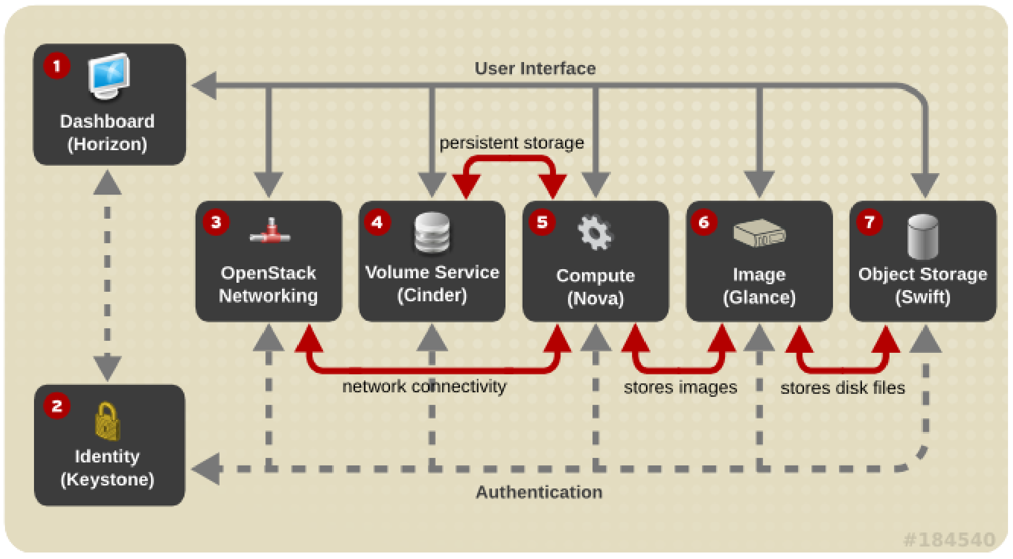


*Figure - 19 OpenStack Architecture*

*i) Horizon :* web browser user interface for creating and managing instances.

*ii) Keystone* : authentication and authorization framework.

*iii) OpenStack networking* : network connectivity as a service.

*iv) Cinder* : persistent block storage for runtime instances.

*v) Nova* : scheduler for networks of virtual machine running on the nodes.

*vi) Glance* : registry for virtual machine images.

*vii) Swift* : file storage and retrieval.

*viii) Ceilometer* : metering engine for collecting billable meters.

*ix) Heat* : orchestration service for template-based virtual machine deployments.

***SWIFT:***

Swift (object) is a service providing the service of Object storage which allows user to store and retrieve files. Swift architecture is distributive architecture to allow horizontal scaling and to provide redundancy as failure-proofing. Data replication is managed by software, allowing greater scalability and redundancy than dedicated hardware. Swift implements its own REST based APIs. Beside this it also implements a middleware which has support for Amazon S3 like APIs.

*Swift Characteristics*

- Swift is an object storage system that is part of the OpenStack project
- Swift is open-source and freely available
- Swift currently powers the largest object storage clouds, including Rackspace Cloud Files, the HP Cloud, IBM Softlayer Cloud and countless private object storage clusters
- Swift can be used as a stand-alone storage system or as part of a cloud compute environment.

- Swift runs on standard Linux distributions and on standard x86 server hardware
- Swift—like Amazon S3—has an eventual consistency architecture, which make it ideal for building massive, highly distributed infrastructures with lots of unstructured data serving global sites.
- All objects (data) stored in Swift have a URL
- Applications store and retrieve data in Swift via an industry-standard RESTful HTTP API
- Objects can have extensive metadata, which can be indexed and searched
- All objects are stored with multiple copies and are replicated in as-unique-as-possible availability zones and/or regions
- Swift is scaled by adding additional nodes, which allows for a cost-effective linear storage expansion
- When adding or replacing hardware, data does not have to be migrated to a new storage system, i.e. there are no fork-lift upgrades
- Failed nodes and drives can be swapped out while the cluster is running with no downtime. New nodes and drives can be added the same way.

Swift is a highly available, distributed, eventually consistent object/blob store. Organizations can use Swift to store lots of data efficiently, safely, and cheaply. Swift can be accessed with HTTP requests directly to the API or by using one of the many Swift client libraries such as Java, Python, Ruby, or JavaScript. This makes it ideal as a primary storage system for data that needs to be stored and accessed via web based clients, devices and applications.

Swift is commonly used with other cloud computing frameworks and as a stand-alone storage system. Swift's open design also enables it to be integrated with enterprise authentication system and IT management tools. Swift's increasing adoption is reflected by how many of the most popular backup and content management applications now support Swift's HTTP API.

## SWIFT ARCHITECTURAL OVERVIEW

A Swift cluster is the distributed storage system used for object storage. It is a collection of machines that are running Swift's server processes and consistency services. Each machine running one or more Swift's processes and services is called a node.

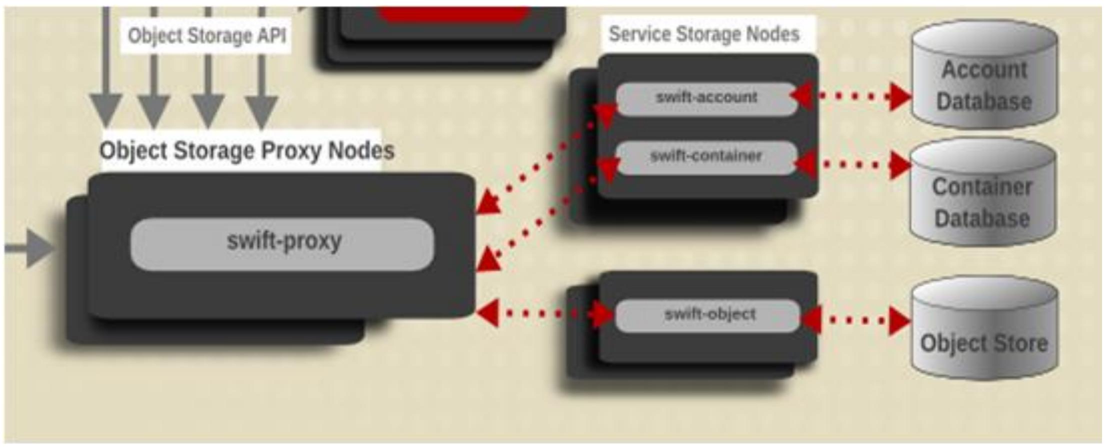


*Figure - 20 Swift Architecture*

Fig. 20 *summarizes the architecture of SWIFT. The Proxy Node acts as a gateway and services request of each user. Auth Node acts as a Authentication Verifier and authenticity of each user is checked. Storage Nodes are the data stores. Swift has the concept of Containers, similar to Buckets in S3. A container is a storage compartment for your data and provides a way for you to organize your data. An object is the basic storage entity and any optional metadata that represents the files you store in the OpenStack Object Storage system. The currently supported API binding for OpenStack Object Storage are: PHP, Python, Java, C#/.NET and Ruby.*

The four Swift server processes are proxy, account, container and object. When a node has only the proxy server process running it is called a proxy node. Nodes running one or more of the other server processes (account, container, or object) will often be called a storage node. Storage nodes contain the data that incoming requests wish to affect, e.g. A PUT request for an object would go to the appropriate nodes

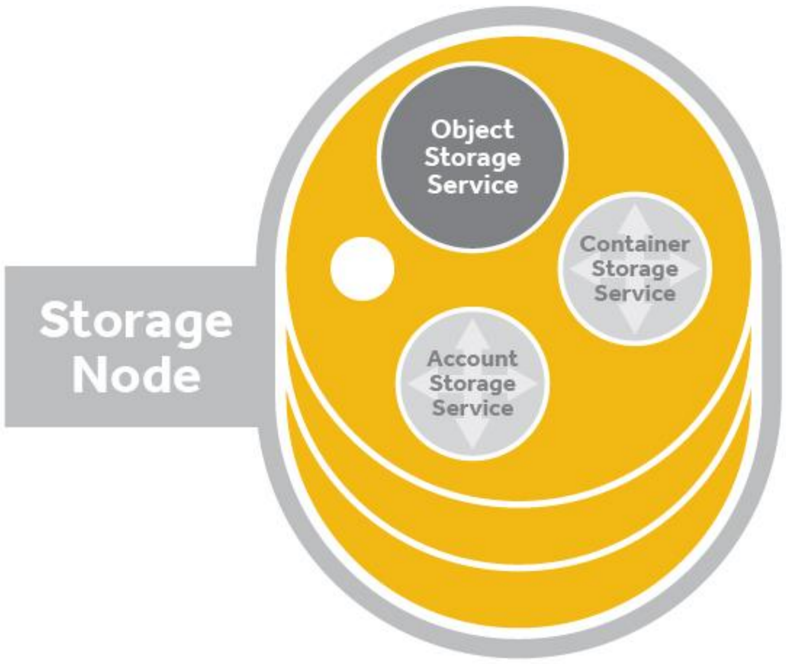


*Figure - 21 Swift Storage node*

running the object server processes. Storage nodes will also have a number of other services running on them to maintain data consistency.

When talking about the same server processes running on the nodes in a cluster we call it the server process layer. e.g., proxy layer, account layer, container layer or object layer.
Let's look a little more closely at the server process layers.

***Server Process Layers***

*Proxy Layer*

The Proxy Server is responsible for tying together the rest of the Swift architecture, as they are the only ones that communicate with external clients due to which they are the first and last to handle an API request. All requests to and responses from the proxy use standard HTTP verbs and response codes.

For each request, it will look up the location of the account, container, or object in the ring and route the request accordingly. For Erasure Code type policies, the Proxy

Server is also responsible for encoding and decoding object data. The public API is also exposed through the Proxy Server.

Proxy servers use a shared-nothing architecture and can be scaled as needed based on projected workloads. A minimum of two proxy servers should be deployed for redundancy. Should one proxy server fail, the others will take over.

A large number of failures are also handled in the Proxy Server. For example, if a valid request is sent to Swift then the proxy server will verify the request, determine the correct storage nodes responsible for the data (based on a hash of the object name) and send the request to those servers concurrently. If one of the primary storage nodes is unavailable, the proxy will choose an appropriate hand-off node to send the request to. The nodes will return a response and the proxy will in turn return the first response (and data if it was requested) to the requester.

The proxy server process looks up multiple locations while retrieving data or responding a request because Swift provides data durability by writing multiple– typically three complete copies of the data and storing them in the system.(Number of replication can be controlled)

*Account Layer*

The account server process handles requests regarding metadata for the individual accounts or the list of the containers within each account. This information is stored by the account server process in SQLite databases on disk.

*Container Layer*

The Container Server's primary job is to handle listings of objects. It doesn't know where those object's are, just what objects are in a specific container. The listings are stored as SQLite database files, and replicated across the cluster similar to how objects are. Also the Container server process handles requests regarding container metadata or the list of objects within each container.

*Object Layer*

The object server process is responsible for the actual storage of objects on the drives of its node, it is basically a very simple blob storage server that can store, retrieve and delete objects stored on local devices. Objects are stored as binary files on the filesystem with metadata stored in the file's extended attributes (xattrs), which means the data and metadata are stored together and copied as a single unit. This requires that the underlying filesystem choice for object servers support xattrs on files. Some filesystems, like ext3, have xattrs turned off by default.

Each object is stored using a path derived from the object name's hash and the operation's timestamp. The timestamp is important as it allows the object server to store multiple versions of an object. Last write always wins, and ensures that the latest object version will be served. A deletion is also treated as a version of the file (a 0 byte file ending with ".ts", which stands for tombstone). This ensures that deleted files are replicated correctly and older versions don't magically reappear due to failure scenarios.

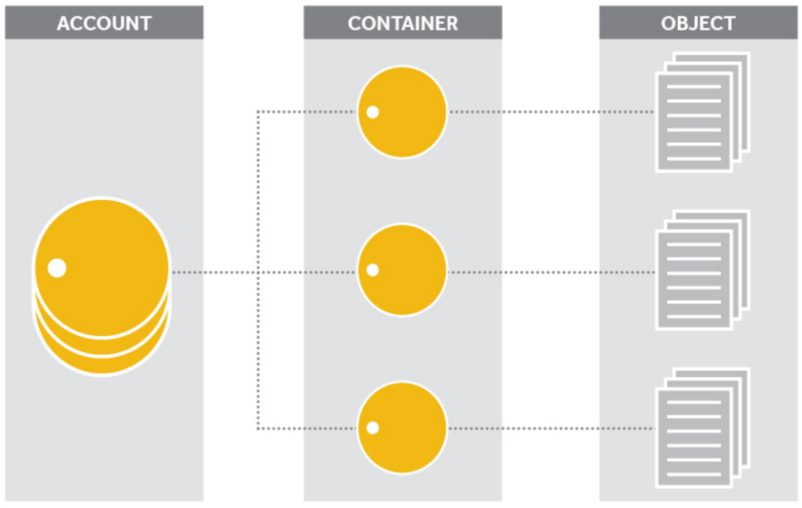


*Figure - 22*

**Consistency Services**

A key aspect of Swift is that it acknowledges that failures happen and is built to work around them. When account, container or object server processes are running on node, it means that data is being stored there. That means consistency services will also be running on those nodes to ensure the integrity and availability of the data.

The two main consistency services are auditors and replicators. There are also a number of specialized services that run in support of individual server process, e.g., the account reaper that runs where account server processes are running.

*Auditors*

Auditors run in the background on every storage node in a Swift cluster and continually scan the disks to ensure that the data stored on disk has not suffered any bit-rot or file system corruption, and the integrity of the objects, containers, and accounts remains. There are account auditors, container auditors and object auditors which run to support their corresponding server process.If an error is found, the file is quarantined, that is the auditor moves the corrupted object to a quarantine area and replication will replace the bad file from another replica. If other errors are found they are logged (for example, an object’s listing can’t be found on any container server it should be).

*Replicators*

Replication is designed to keep the system in a consistent state in the face of temporary error conditions like network outages or drive failures.

Account, container, and object replicator processes run in the background on all nodes that are running the corresponding services. The replication processes ensure they all contain the latest version by examine its local node and compare the accounts, containers, or objects against the copies on other nodes in the cluster. If one of other nodes has an old or missing copy, then the replicator will send a copy of its local data out to that node.

Object replication uses a hash list to quickly compare subsections of each partition, and container and account replication use a combination of hashes and shared high water marks.

Replication updates are push based ,they only push their local data out to other nodes; they do not pull in remote copies in if their local data is missing or out of date.

For object replication, updating is just a matter of re-syncing files to the peer. Account and container replication push missing records over HTTP or re-sync whole database files.

The replicator also handles object and container deletions. Object deletion starts by creating a zero-byte tombstone file that is the latest version of the object. This version is then replicated to the other nodes and the object is removed from the entire system. Container deletion can only happen with an empty container. It will be marked as deleted and the replicators push this version out.

*Updater*

There are times when container or account data cannot be immediately updated. This usually occurs during failure scenarios or periods of high load. If an update fails, the updater then takes over and will process the failed updates. This is where an eventual consistency window will most likely come in to play. For example, suppose a container server is under load and a new object is put in to the system. The object will be immediately available for reads as soon as the proxy server responds to the client with success. However, the container server did not update the object listing, and so the update would be queued for a later update. Container listings, therefore, may not immediately contain the object.

Swift have two updaters Container and Object Updaters :

The container updater service runs to support accounts, it will update the container listings in the accounts and account metadata which contains object count, container count and bytes used.

The object updater runs to support containers, but as a redundant service. The object server process is the primary updater. Only if it fails with an update attempt will the object updater take over and then update the object listing in the containers and container metadata i.e. the object count and bytes used.

*Account Reaper*

The Account Reaper removes data from deleted accounts in the background. When an the service makes its rounds on a node and finds an account marked as deleted, it starts stripping out all objects and containers associated with the account. With each pass it will continue to dismantle the account until it is emptied and removed. The reaper has a delay value that can be configured so the reaper will wait before it starts deleting data—this is used to guard against erroneous deletions.

## Cluster Architecture

*Nodes*

A node is a machine that is running one or Swift processes. When there are multiple nodes running that provide all the processes needed for Swift to act as a distributed storage system they are considered to be a cluster.

Within a cluster the nodes will also belong to two logical groups: regions and nodes. Regions and nodes are user-defined and identify unique characteristics about a collection of nodes-- usually geography location and points of failure, such as all the power running to one rack of nodes. These ensure Swift can place data across different parts of the cluster to reduce risk.

*Regions*

Regions are user-defined and usually indicate when parts of the cluster are physically separate --usually a geographical boundary. A cluster has a minimum of one region and there are many single region clusters as a result. A cluster that is using two or more regions is a multi-region cluster.

When a read request is made, the proxy layer favors nearby copies of the data as measured by latency. When a write request is made the proxy layer, by default, writes to all the locations simultaneously. There is an option called write affinity that when enabled allows the cluster to write all copies locally and then transfer them asynchronously to the other regions.

*Zones*

Within regions, Swift allows allows availability zones to be configured to isolate failure boundaries. An availability zone should be defined by a distinct set of physical hardware whose failure would be isolated from other zones. In a large deployment, availability zones may be defined as unique facilities in a large data center campus. In a single datacenter deployment, the availability zones may be different racks. While there does need to be at least one zone in a cluster, it is far more common for a cluster to have many zones.

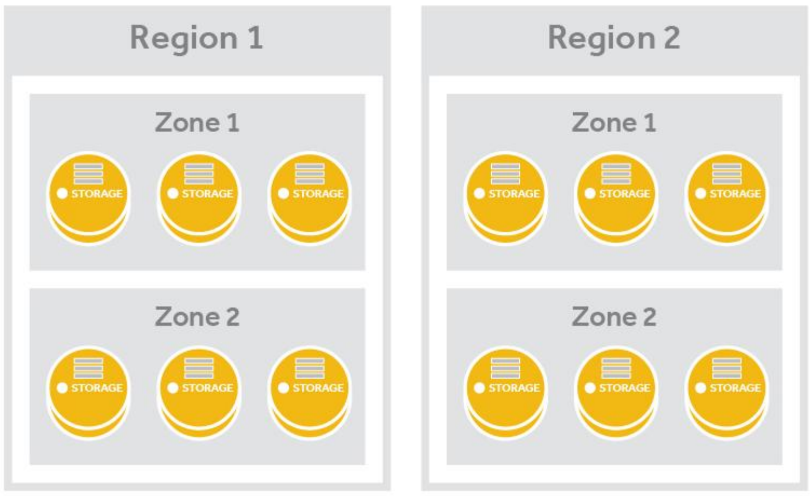


*Figure - 23 Storage zones can be deployed across geographic regions.*

**Rings**

Swift supports Distributed Hashed Table(DHTs) called Rings which represents a mapping between the names of entities stored on disk and their physical location. There are separate rings for accounts, containers, and one object ring per storage policy. When other components need to perform any operation on an object, container, or account, they need to interact with the appropriate ring to determine its location in the cluster.

The Ring is used by the Proxy server and several background processes (like replication). It maintains this mapping using zones, devices, partitions (a partition is just a directory sitting on a disk with a corresponding hash table of what it contains) and replicas. Each partition in the ring is replicated, by default, 3 times across the cluster, and the locations for a partition are stored in the mapping maintained by the ring. The ring is also responsible for determining which devices are used for handoff in failure scenarios. The replicas of each partition will be isolated onto as many distinct regions, zones, servers and devices as the capacity of these failure domains allow.

Data is evenly distributed across the capacity available in the cluster as described by the devices weight. Weights can be used to balance the distribution of partitions on drives across the cluster. This can be useful, for example, when different sized drives are used in a cluster. Device weights can also be used when adding or removing capacity or failure domains to control how many partitions are reassigned during a rebalance to be moved as soon as replication bandwidth allows.

When partitions need to be moved around (for example if a device is added to the cluster), the ring ensures that a minimum number of partitions are moved at a time, and only one replica of a partition is moved at a time.

# 5. User Tasks

1. *Login and Signup:* To use this system login and sign-up is very essential. A user can log in and then generate a password for future use.
2. *Setup Cluster*: Here the user needs to setup the cluster based on his/her policies including RAM, Storage, nodes and security.
3. *User Jobs*: Here according to the selection of data by the user he/she will be guided through the type of queering the client wants to carry out.
    1. Sorting plans
    2. Word counts

# CONCLUSION

This Tool helps the users to manage Hadoop with the help of web interface. Connecting with other enthusiasts and exchanging suggestions with them help everyone to access the latest techniques and trends being followed. With the help of this tool the aim is to look forward to build up a world through the boon of Big Data Analytics using Hadoop and  OpenStack Swift.

# REFERENCES

[1] www.apache.com

[2] www.pigonthecall.com

[3] www.msdn.microsoft.com

[4] www.stackoverflow.com

[5] www.bigdatauniversity.com

[6] www.apache.org

[7] http://en.wikipedia.org/wiki/OpenStack

[8] Hadoop in Practice by *Alex Homes*

[9] Hadoop operations by *Eric Sammer*

[10] Hadoop the definitive guide by *O'reilly*

[11] http://docs.openstack.org/openstack-ops/content/upstream_openstack.html

[12] CL-120, Red Hat OpenStack Administration Red Hat OpenStack 4.0, Student Workbook by Red Hat

[13] Red Hat OpenStack Adminstration Red Hat OpenStack Release en-1-20140207 Student workbook.

[14] https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux_OpenStack_Platform/5/html/Installation_and_Configuration_Guide/Architecture6.html.

[15] https://swiftstack.com/openstack-swift/

[16] http://docs.openstack.org/developer/swift/

# APPENDIX

*PROJECT CODE*

*i)Web GUI*

```
<html>

<head>
      <title>Hadoop Home</title>
      <link rel="stylesheet" type="text/css" href="style.css" />
</head>

<body >

      <div class="wrapper">
      <img src="logo.jpg">

      <ul>
      <li><a href="">Hadoop Home</a>
      <div>

<ul>
      <li> <a href="">Configure Policy!</a></li>
      <li> <a href="">Scan for customer</a></li>
      <li> <a href="">Enter customer</a></li>

</ul>
      </div>
      </li>
            <li><a href="">Setup Hadoop version 1</a></li>
            <li><a href="">Setup Hadoop version 2</a></li>
            <li><a href="">MRv1</a></li>
            <li><a href="">MRv2</a></li>
      </ul>
      </div>

</body></html>
```

***ii)Python script***

***Scanning the system.***

```
#!/usr/bin/python2

import commands

print "content-type:text/html; charset=utf-8\r\n"

print "<html>"
print "<head>"
print "<title>Scan output</title>"
l=commands.getoutput('nmap  -v 192.168.20.0-255 -p 22 | grep open |
grep Dis | cut -f6 -d" " ')


#print commands.getoutput('chown -R apache:apache /var/www/cgi-
bin/files')
#print commands.getoutput('chmod u+w /var/www/cgi-bin/files -R')

f=open("/var/www/cgi-bin/files/iplist.txt", "w")
f.write(l)
f.close()


fi=open("/var/www/cgi-bin/files/iplist.txt", "r")

print "<h4>Select node</h4>"

print "<form action='check.py' method='get'>"

for i in fi:
      print i+"\t <input type='checkbox' name='ip' value='%s' />" %i,
      print "<br>"

print "<input type='submit' value='Check' />"
print "</form>"
```

```
fi.close()


print "</body>"
print "</html>"
```

***Checking the system***

```
#!/usr/bin/python2



import commands
import cgi, sys, time
print "content-type:text/html\r\n"

print "<html>"
print "<head>"
print "<title>Checking</title>"
print "</head>"
print "<body>"


print "<h2>Summary</h2>"

ip=cgi.FormContent()['ip']



print"<h4>Select a name node</h4>"

print "<form action='nnode.py' method='get' >"
for i in ip:
      i=i.strip()
      print "<input type='checkbox' name='node' value='%s' />" %i
      print i
      print "<br>"
      print "Free RAM:"
      m=commands.getoutput("sudo sshpass -p redhat ssh -o 
StrictHostKeyChecking=no root@%s free -m | head -2 | grep Mem | awk 
'{print $4}'  " %i)
      print m,"MB"
      print
      print "<br>"

      print "Free HD:"
      r=commands.getoutput("sudo sshpass -p redhat ssh -o 
StrictHostKeyChecking=no root@%s  df -hT | head -2 | grep / | awk 
'{print $5}'" %i)
      t=commands.getoutput("sudo sshpass -p redhat ssh -o 
StrictHostKeyChecking=no root@%s df -hT | grep tmpfs | awk '{print 
$4}' " %i)

      #r=int(r)
      #t=int(t)
      #r=r-(t/(1024*1024))
      print r
      print "<br>"

      print "CPU cores: "
      c = commands.getoutput("sudo sshpass -p redhat ssh -o 
StrictHostKeyChecking=no root@%s cat /proc/cpuinfo | grep 'cpu core' 
| head -1 | awk '{print $4}'" %i)
      print c
      print "<br>"

print "<input type='submit' value='Establish node' />"

print "</body>"
print "</html>"
```

## *Establishing the hdfs  nodes*

```
#!/usr/bin/python2

import commands
import cgi, sys, time
print "content-type:text/html\r\n"

print "<html>"
print "<head>"
print "<title>Establishing Namenode</title>"
print "</head>"
print "<body>"
f=open("/var/www/cgi-bin/files/nnode.txt", "w")
nn=cgi.FormContent()['node']
for i in nn:
	i=i.strip()

	ins=commands.getoutput("sudo sshpass -p redhat ssh -o 
StrictHostKeyChecking=no root@%s yum install hadoop -y " %i)

	jdk=commands.getoutput("sudo sshpass -p redhat ssh -o 
StrictHostKeyChecking=no root@%s yum install jdk -y " %i)

	print "<p>Hadoop and jdk is installed for name node </p>"
	f.write(i + "\n")

f.close()

print "<form action='scan1.py' method='post'>"
print "<input type='submit' value='Continue' />"

print "</body>"
print "</html>"
```

```
#!/usr/bin/python2

import commands
import cgi

print "content-type:text/html; charset=utf-8\r\n"

print "<html>"
print "<head>"
print "<title>Installing</title>"

print"</head>"
print "<body>"

x=cgi.FormContent()['h'][0]

dn=cgi.FormContent()['dn1']

fi=open("/var/www/cgi-bin/files/dnode.txt", "w")

for ip in dn:
      fi.write(ip+"\n")

fi.close()

fi=open("/var/www/cgi-bin/files/dnode.txt", "r")

for i in fi:
      i=i.strip()
      print "<p>Installing hadoop</p>"
      ins=commands.getoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s yum install hadoop -y " %i)

      print "<p>Hadoop is installed for "+i+" </p>"
      jdk=commands.getoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s yum install jdk -y " %i)

      print "<p>JDK is installed for "+i+" </p>"

fi.close
```

```
print "<form action='scpfile.py' method='post' >"
print"<p>Click to continue with the Configuration process.</p>"
print "<input type='submit' value='Continue' />"
print "</form>"

print "</body>"
print "</html>"
```

### *Partitioning the local disk for HDFS*

```
#!/usr/bin/python2



import commands
print "content-type:text/html\r\n"

print "<html>"
print "<head>"
print "<title>Checking</title>"
print "</head>"
print "<body>"

f=open("/var/www/cgi-bin/files/nnode.txt", "r")

for i in f:
      i=i.strip()
      chk=0
      m=commands.getoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s parted /dev/sda print free | grep
'Free Space' | grep GB " %i)
      m=m.split()
      s=m[0].split('G')
      S=s[0]
      S=int(S)
      E=S+1
      if E<m[1]:
            S=str(S)
            E=str(E)
            p=commands.getoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s parted /dev/sda mkpartfs logical
ext2 %sGB %sGB " %(i, S, E))
            commands.getoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s partx -a /dev/sda" %i)
            commands.getoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s partx -a /dev/sda" %i)
            commands.getoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s partx -a /dev/sda" %i)
            d=commands.getoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s fdisk -cul /dev/sda | tail -1 | awk
'{print $1}'" %i)
            commands.getoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s mkdir /media/name" %i)
            commands.getoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s mount %s /media/name" %(i,d))
            chk=1

      if chk==1:
            print "Partition created successfully"
            print "<br>"
      else:
            print "Error"



f.close()

f=open("/var/www/cgi-bin/files/dnode.txt", "r")

for i in f:
      i=i.strip()
      chk=0
      print i
      print "<input type='checkbox' name='node' value='%s' />" %i
      l=commands.getoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s parted /dev/sdb print free | grep
'Free Space' | grep MB " %i)
      l=l.split()
      st=l[0].split('k')
      St=st[0]
      St=int(St)
      Ed=St+1782579
      print Ed
      if Ed<l[1]:
            St=str(St)
            Ed=str(Ed)
            p=commands.getoutput("sudo sshpass -p redhat ssh -o 
StrictHostKeyChecking=no root@%s parted /dev/sdb mkpartfs logical 
ext2 %skB %skB Ignore " %(i,St,Ed))
            commands.getoutput("sudo sshpass -p redhat ssh -o 
StrictHostKeyChecking=no root@%s partx -a /dev/sdb" %i)
            commands.getoutput("sudo sshpass -p redhat ssh -o 
StrictHostKeyChecking=no root@%s partx -a /dev/sdb" %i)
            commands.getoutput("sudo sshpass -p redhat ssh -o 
StrictHostKeyChecking=no root@%s partx -a /dev/sdb" %i)
            d=commands.getoutput("sudo sshpass -p redhat ssh -o 
StrictHostKeyChecking=no root@%s fdisk -cul /dev/sdb | tail -1 | awk 
'{print $1}'" %i)
            commands.getoutput("sudo sshpass -p redhat ssh -o 
StrictHostKeyChecking=no root@%s mkdir /media/data" %i)
            commands.getoutput("sudo sshpass -p redhat ssh -o 
StrictHostKeyChecking=no root@%s mount %s /media/data" %(i,d))
            chk=1

      if chk==1:
            print "Partition created successfully"
            print "<br>"
      else:
            print "Error"
f.close()

print "<form action='hadoopinstall.py' method='post' >"
print "<input type='submit' value='Continue' />"

print "</body>"
print "</html>"
```

## *Configuring the HDFS nodes*

```
#!/usr/bin/python2

import commands
import cgi, sys, time
print "content-type:text/html\r\n"

print "<html>"
print "<head>"
print "<title>Tranferring files</title>"
print "</head>"
print "<body>"

f=open("/var/www/cgi-bin/files/nnode.txt", "r")

for i in f:
      i=i.strip()
      print i
f.close()

f=open("/var/www/cgi-bin/files/core-site.xml", "a")
f.write("<configuration>\n")
f.write("<property>\n")
f.write("<name>fs.default.name</name>\n")
f.write("<value>hdfs://"+i+":10001</value>\n")
f.write("</property>\n")
f.write("</configuration>\n")
f.close()

f=open("/var/www/cgi-bin/files/nnode.txt", "r")

for i in f:
      i=i.strip()
      commands.getstatusoutput("sudo sshpass -p redhat scp
/var/www/cgi-bin/files/core-site.xml root@%s:/etc/hadoop/ " %i)
      commands.getstatusoutput("sudo sshpass -p redhat scp
/var/www/cgi-bin/files/hadoop-env.sh root@%s:/etc/hadoop/ " %i)
```

```
        commands.getstatusoutput("sudo sshpass -p redhat scp
/var/www/cgi-bin/files/n/hdfs-site.xml root@%s:/etc/hadoop/ " %i)
        commands.getstatusoutput("sudo sshpass -p redhat scp
/root/.bash_profile root@%s:/root/ " %i)
        commands.getstatusoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s . /root/.bash_profile " %i)
f.close()

fi=open("/var/www/cgi-bin/files/dnode.txt", "r")
for i in fi:
        i=i.strip()
        commands.getstatusoutput("sudo sshpass -p redhat scp
/var/www/cgi-bin/files/core-site.xml root@%s:/etc/hadoop/ " %i)
        commands.getstatusoutput("sudo sshpass -p redhat scp
/var/www/cgi-bin/files/hadoop-env.sh root@%s:/etc/hadoop/ " %i)
        commands.getstatusoutput("sudo sshpass -p redhat scp
/var/www/cgi-bin/files/hdfs-site.xml root@%s:/etc/hadoop/ " %i)
        commands.getstatusoutput("sudo sshpass -p redhat scp
/root/.bash_profile root@%s:/root/ " %i)
        commands.getstatusoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s source /root/.bash_profile " %i)
fi.close()

print "Congratulation Hadoop has configured for your cluster"
print " Now you may proceed to start services "

print "<form method='post' action='uploaddata.py' >"
print "<input type='submit' value='Start Service' />"
print "</form>"
print "</body>"
print "</html>"
```

### *Starting namenodes and datanodes*

```
#!/usr/bin/python2

import commands
print "content-type:text/html\r\n"
print "<html>"
print "<head>"
print "<title>Uploading data</title>"
print "</head>"
print "<body>"

f=open("/var/www/cgi-bin/files/nnode.txt", "r")

print "sakdgjhsad"
for i in f:
      i=i.strip()
      #commands.getstatusoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no " %i)
      commands.getstatusoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s hadoop-daemon.sh start namenode "
%i)

f.close()

print "KASDGJGSDJHGD"
fi=open("/var/www/cgi-bin/files/dnode.txt", "r")
for i in fi:
      i=i.strip()
      commands.getstatusoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s hadoop-daemon.sh start datanode "
%i)
fi.close()



f=open("/var/www/cgi-bin/files/nnode.txt", "r")

print "sakdgjhsad"
for i in f:
      i=i.strip()
      commands.getstatusoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s hadoop fs -mkdir /input" %i)
      commands.getstatusoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s hadoop fs -chown apache:apache
/input" %i)
```

```
      commands.getstatusoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s hadoop fs -chmod 777 /input" %i)
      print "<p>Directory is made in the hdfs</p>"
      #commands.getstatusoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s hadoop fs -put /root/Desktop/b.txt
/input/" %i)
      print "Dahdj"

f.close()


#print "<form enctype='multipart/form-data' action='upload.py'
method='post'>
#print "Select a File to upload: <input type='file' name='f'/>"
#print "<input type='submit' value='Upload' />"
##print "</from>"

print "<p>Services are started please upload data"
print "<form method='post' action='/html/upload.html' >"
print "<input type='submit' value='Home' />"
print "</form>"
print "</body>"
print "</html>"
```

## *Establishing MapReduce Node*

```
#!/usr/bin/python2

import commands
import cgi, sys, time
print "content-type:text/html\r\n"

print "<html>"
print "<head>"
print "<title>Establishing JobTracker</title>"
print "</head>"
print "<body>"
f=open("/var/www/cgi-bin/files/jt.txt", "w")
```

```
nn=cgi.FormContent()['node']
for i in nn:
	i=i.strip()
	print "<p>Configuring Mapred framework</p>"
	f.write(i + "\n")

f.close()



f=open("/var/www/cgi-bin/files/jt.txt", "r")

for i in f:
	i=i.strip()
	ins1=commands.getoutput("sudo sshpass -p redhat ssh -o 
StrictHostKeyChecking=no root@%s yum install hadoop -y " %i)

	print "shas"
	jdk1=commands.getoutput("sudo sshpass -p redhat ssh -o 
StrictHostKeyChecking=no root@%s yum install jdk -y " %i)

	print "jdk installed"

f.close()

print "<p>Continue to establish tasktrackers</p>"
print "<form action='mapred1.py' method='post'>"
print "<input type='submit' value='Continue' />"

print "</body>"
print "</html>"
```

### *Configuring the MapReduce framework nodes:*

```
#!/usr/bin/python2

import commands
import cgi, sys, time
print "content-type:text/html\r\n"
print "<html>"
print "<head>"
print "<title>Uploading data</title>"
print "</head>"
print "<body>"

fi=open("/var/www/cgi-bin/files/jt.txt", "r")

for i in fi:
	i=i.strip()
	f=open("/var/www/cgi-bin/files/mapred-site.xml", "a")
	f.write("<configuration>\n")
	f.write("<property>\n")
	f.write("<name>mapred.job.tracker</name>\n")
	f.write("<value>"+i+":9001</value>\n")
	f.write("</property>\n")
	f.write("</configuration>\n")
	f.close()

fi.close()

fi=open("/var/www/cgi-bin/files/jt.txt", "r")



print "sakdgjhsad"
for i in fi:
	i=i.strip()

	commands.getoutput("sudo sshpass -p redhat scp /var/www/cgi-
bin/files/mapred-site.xml root@%s:/etc/hadoop/ " %i)
	commands.getoutput("sudo sshpass -p redhat scp /var/www/cgi-
bin/files/core-site.xml root@%s:/etc/hadoop/ " %i)
	commands.getoutput("sudo sshpass -p redhat scp
/root/.bash_profile root@%s:/root/ " %i)
	commands.getoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s source /root/.bash_profile " %i)
	#commands.getstatusoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s hadoop-daemon.sh start jobtracker"
%i)
```

```
fi.close()

fi=open("/var/www/cgi-bin/files/tt.txt", "r")
for i in fi:
      i=i.strip()

      commands.getstatusoutput("sudo sshpass -p redhat scp
/var/www/cgi-bin/files/mapred-site.xml root@%s:/etc/hadoop/ " %i)

      commands.getstatusoutput("sudo sshpass -p redhat scp
/var/www/cgi-bin/files/core-site.xml root@%s:/etc/hadoop/ " %i)

      commands.getstatusoutput("sudo sshpass -p redhat scp
/root/.bash_profile root@%s:/root/ " %i)
      commands.getstatusoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s source /root/.bash_profile " %i)
      #commands.getstatusoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s hadoop-daemon.sh start tasktracker"
%i)
fi.close()

print "everything done"

print "<p>Continue to start services</p>"
print "<form action='start.py' method='post'>"
print "<input type='submit' value='Continue' />"
print "</form>"
print "</body>"
print "</html>"
```

### *Starting JobTracker and TaskTrackers*

```
#!/usr/bin/python2

import commands
import cgi, sys, time
print "content-type:text/html\r\n"

print "<html>"
print "<head>"
print "<title>Uploading data</title>"
print "</head>"
print "<body>"

fi=open("/var/www/cgi-bin/files/jt.txt", "r")
print "sakdgjhsad"
for i in fi:
      i=i.strip()
      commands.getoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s hadoop-daemon.sh start jobtracker"
%i)

fi.close()

print "KASDGJGSDJHGD"
f=open("/var/www/cgi-bin/files/tt.txt", "r")
for i in f:
      i=i.strip()
      commands.getoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s hadoop-daemon.sh start tasktracker"
%i)
f.close()

n=open("/var/www/cgi-bin/files/nnode.txt", "r")
for i in n:
      i=i.strip()
      #commands.getoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s hadoop fs -put /root/Desktop/b.txt
/input" %i)
n.close()

print "everything done"

print "<p>Continue to start</p>"
print "<form action='start.py' method='post'>"
print "<input type='submit' value='Continue' />"
print "</form>"
print "</body>"
print "</html>
```

***Uploading Data and Running Job***

```
<html>
<body>
<form enctype="multipart/form-data" action="upload.py" method="post">
<p>File: <input type="file" name="f"></p>
<p><input type="submit" value="Upload"></p>
</form>
</body>
</html>

#!/usr/bin/python2

import commands
import cgi, sys, time
print "content-type:text/html\r\n"

print "<html>"
print "<head>"
print "<title>Uploading data</title>"
print "</head>"
print "<body>"
form = cgi.FieldStorage()
fn=form['f'].filename
print fn
data=form['f'].file.read()
ff=open("/var/www/cgi-bin/files/"+fn, "w")
ff.write(data)
ff.close()

fi=open("/var/www/cgi-bin/files/nnode.txt", "r")
print "sakdgjhsad"
for i in fi:
      i=i.strip()
      commands.getoutput("sudo sshpass -p redhat scp /var/www/cgi-
bin/files/%s root@%s:/root/Desktop/" %(fn,i))
      commands.getoutput("sudo sshpass -p redhat ssh -o
StrictHostKeyChecking=no root@%s hadoop fs -put /root/Desktop/%s
/input/%s" %(i,fn,fn))
fi.close()
```

```
print "file is uploaded in hdfs"

print "<form action='index.py' method='post'>"
print "<input type='submit' value='Home' />"
print "</form>"

print "<form action='job.py' method='post'>"
print "<input type='submit' value='Run job' />"
print "</form>"
print "</body>"
print "</html>"
```