Working with Big Data: Google File System, Hadoop Distributed File System (HDFS) – Building blocks of Hadoop (Namenode, Datanode, Secondary Namenode, JobTracker, TaskTracker), Introducing and Configuring Hadoop cluster (Local, Pseudo-distributed mode, Fully Distributed mode), Configuring XML files.

FILE SYSTEM

A file system is used to control how data is stored and retrieved. Without a file system, information placed in a storage area would be one large body of data with no way to tell where one piece of information stops and the next begins. By separating the data into pieces and giving each piece a name, the information is easily isolated and identified. Taking its name from the way paper-based information systems are named, each group of data is called a "file". The structure and logic rules used to manage the groups of information is called a "file system". The file system manages access to both the content of files and the metadata about those files. It is responsible for arranging storage space; reliability, efficiency, and tuning with regard to the physical storage medium are important design considerations.

DISTRIBUTED FILE SYSTEM (DFS)

A distributed file system is a client/server-based application that allows clients to access and process data stored on the server as if it were on their own computer. When a user accesses a file on the server, the server sends the user a copy of the file, which is cached on the user's computer while the data is being processed and is then returned to the server.
DFS is a mechanism for sharing files. Distributed file system (DFS) is used to make files distributed across multiple servers appear to users as if they reside in one place on the network.

BENEFITS OF DFS

Resources management and accessibility (users access all resources through a single point)
Accessibility(users do not need to know the physical location of the shared folder)
Fault tolerance (shares can be replicated)
Workload management (DFS allows administrators to distribute shared folders and workloads across several servers for more efficient network and server resources use), and many others.

2.1 GOOGLE FILE SYSTEM

GFS is a proprietary distributed file system developed by Google for its own use. It is designed to provide efficient, reliable access to data using large clusters of commodity hardware. Commodity hardware, in an IT context, is a device or device component that is relatively inexpensive, widely available and more or less interchangeable with other hardware of its type.
A new version of the Google File System is codenamed Colossus which was released in 2010.
GFS was implemented especially for meeting the rapidly growing demands of Google's data processing needs.

2.1.1 TRADITIONAL DESIGN ISSUES OF GFS

- Performance
- Scalability
- Reliability
- Availability

2.1.2 DIFFERENT POINTS IN GFS DESIGN

- Component failures are the norm rather than the exception
  - ❖ Constant monitoring, error detection, fault tolerance and automatic recovery must be integral to the system.
- Files are huge that multi-GB files are common
  - ❖ Design assumptions and parameters such as I/O operations and block sizes have to be revisited.
- Most files are mutated by appending new data rather than overwriting existing data

- appending becomes the focus of performance optimization and atomicity guarantees, while caching data blocks in the client loses its appeal.

2.1.3 GFS DESIGN

GFS is enhanced for Google's core data storage and usage needs (primarily the search engine which can generate enormous amounts of data that needs to be retained.

2.1.4 INTERFACE

GFS provides a familiar file system interface, though it does not implement a standard API such as POSIX. Files are organized hierarchically in directories and identified by pathnames. GFS support the usual operations to create, delete, open, close, read, and write files. Moreover, GFS has snapshot and record append operations. Snapshot creates a copy of a file or a directory tree at low cost. Record append allows multiple clients to append data to the same file concurrently while guaranteeing the atomicity of each individual client's append. It is useful for implementing multi-way merge results and producer consumer queues that many clients can simultaneously append to without additional locking. We have found these types of files to be invaluable in building large distributed applications.

2.1.5 ARCHITECTURE

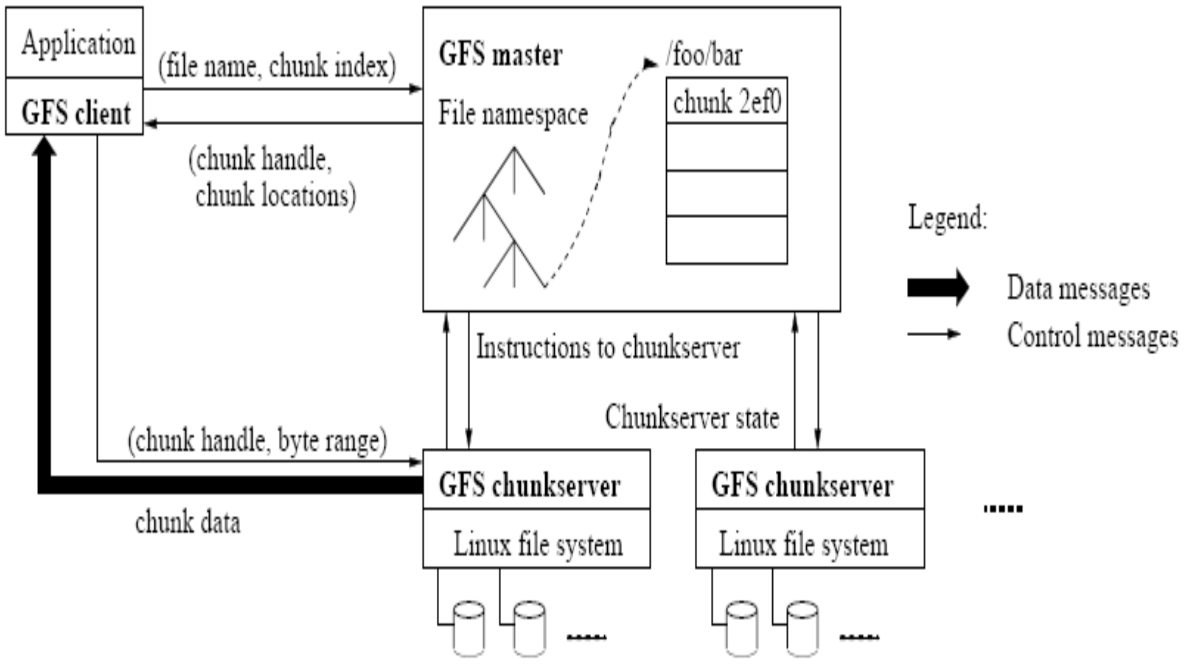


Figure 1: GFS Architecture

A GFS cluster consists of a single master and multiple chunkservers and is accessed by multiple clients, as shown in Figure 1. Each of these is typically a commodity Linux machine running a user-level server process. It is easy to run both a chunkserver and a client on the same machine, as long as machine resources permit and the lower reliability caused by running possibly flaky application code is acceptable.

Files are divided into fixed-size chunks. Each chunk is identified by an immutable and globally unique 64 bit chunk handle assigned by the master at the time of chunk creation. Chunk servers store chunks on local disks as Linux files and read or write chunk data specified by a chunk handle and byte range. For reliability, each chunk is replicated on multiple chunk servers. By default, three replicas will be stored, though users can designate different replication levels for different regions of the file namespace. The master maintains all file system metadata. This includes the namespace, access control information, the mapping from files to chunks, and the

current locations of chunks. It also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunk servers.
The master periodically communicates with each chunk server in HeartBeat messages to give it instructions and collect its state. GFS client code linked into each application implements the file system API and communicates with the master and chunk servers to read or write data on behalf of the application. Clients interact with the master for metadata operations, but all data-bearing communication goes directly to the chunk servers. Neither the client nor the chunk server caches file data. Client caches offer little benefit because most applications stream through huge files or have working sets too large to be cached. Not having them simplifies the client and the overall system by eliminating cache coherence issues. (Clients do cache metadata, however.) Chunk servers need not cache file data because chunks are stored as local files and so Linux's buffer cache already keeps frequently accessed data in memory.

The building blocks of Hadoop

On a fully configured cluster, "running Hadoop" means running a set of daemons, or resident programs, on the different servers in our network. These daemons have specific roles; some exist only on one server, some exist across multiple servers. The daemons include
NameNode
DataNode
Secondary NameNode
JobTracker
TaskTracker

NameNode

Hadoop employs a master/slave architecture for both distributed storage and distributed computation. The distributed storage system is called the Hadoop File System , or HDFS. The NameNode is the master of HDFS that directs the slave DataNode daemons to perform the low-level I/O tasks. The NameNode is the bookkeeper of HDFS; it keeps track of how your files are broken down into file blocks, which nodes store those blocks, and the overall health of the distributed filesystem.
The function of the NameNode is memory and I/O intensive. As such, the server hosting the NameNode typically doesn't store any user data or perform any computations for a MapReduce program to lower the workload on the machine.
There is unfortunately a negative aspect to the importance of the NameNode—it's a single point of failure of your Hadoop cluster. For any of the other daemons, if their host nodes fail for software or hardware reasons, the Hadoop cluster will likely continue to function smoothly or you can quickly restart it. Not so for the NameNode.

DataNode

Each slave machine in your cluster will host a DataNode daemon to perform the grunt work of the distributed filesystem—reading and writing HDFS blocks to actual files on the local filesystem. When you want to read or write a HDFS file, the file is broken into blocks and the NameNode will tell your client which DataNode each block resides in. Your client communicates directly with the DataNode daemons to process the local files corresponding to the blocks. Furthermore, a DataNode may communicate with other DataNodes to replicate its data blocks for redundancy.
The following figure illustrates the roles of the NameNode and DataNodes. In this figure, we show two data files, one at /user/chuck/data1 and another at /user/james/data2. The data1 file takes up three blocks, which we denote 1, 2, and 3, and the data2 file consists of blocks 4 and 5. The content of the files are distributed among the DataNodes. In this illustration, each block has three replicas. For example, block 1 (used for data1) is replicated over the three rightmost DataNodes. This ensures that if any one DataNode crashes or becomes inaccessible over the network, you'll still be able to read the files.
DataNodes are constantly reporting to the NameNode. Upon initialization, each of the DataNodes informs the NameNode of the blocks it's currently storing. After this mapping is complete, the DataNodes continually poll

the NameNode to provide information regarding local changes as well as receive instructions to create, move, or delete blocks from the local disk.

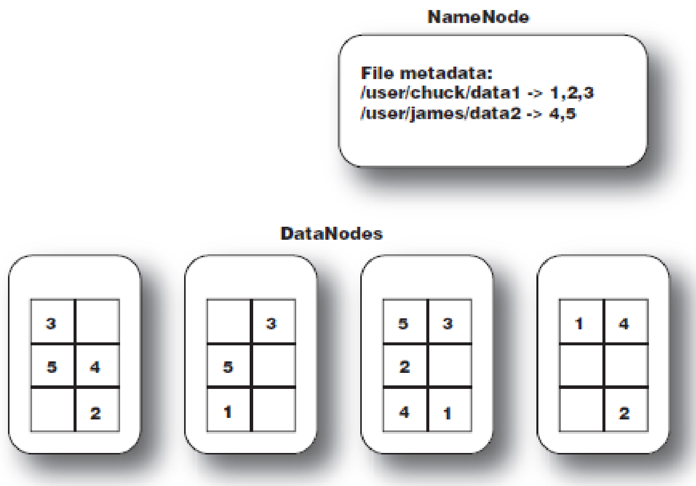


Figure NameNode /DataNode interaction in HDFS.

The NameNode keeps track of the file metadata—which files are in the system and how each file is broken down into blocks. The DataNodes provide backup store of the blocks and constantly report to the NameNode to keep the metadata current.

Secondary NameNode

The Secondary NameNode (SNN) is an assistant daemon for monitoring the state of the cluster HDFS. Like the NameNode, each cluster has one SNN, and it typically resides on its own machine as well. No other DataNode or TaskTracker daemons run on the same server. The SNN differs from the NameNode in that this process doesn't receive or record any real-time changes to HDFS. Instead, it communicates with the NameNode to take snapshots of the HDFS metadata at intervals defined by the cluster configuration.
The NameNode is a single point of failure for a Hadoop cluster, and the SNN snapshots help minimize the downtime and loss of data. Nevertheless, a NameNode failure requires human intervention to reconfigure the cluster to use the SNN as the primary NameNode.

JobTracker

The JobTracker daemon is the liaison between your application and Hadoop. Once you submit your code to your cluster, the JobTracker determines the execution plan by determining which files to process, assigns nodes to different tasks, and monitors all tasks as they're running. Should a task fail, the JobTracker will automatically relaunch the task, possibly on a different node, up to a predefined limit of retries. There is only one JobTracker daemon per Hadoop cluster. It's typically run on a server as a master node of the cluster.

TaskTracker

the JobTracker is the master for the overall execution of a MapReduce job and the TaskTrackers manage the execution of individual tasks on each slave node. The following figure shows this interaction.
Each TaskTracker is responsible for executing the individual tasks that the JobTracker assigns. Although there is a single TaskTracker per slave node, each TaskTracker can spawn multiple JVMs to handle many map or reduce tasks in parallel. One responsibility of the TaskTracker is to constantly communicate with the JobTracker. If the JobTracker fails to receive a heartbeat from a TaskTracker within a specified amount of time, it will assume the TaskTracker has crashed and will resubmit the corresponding tasks to other nodes in the cluster.

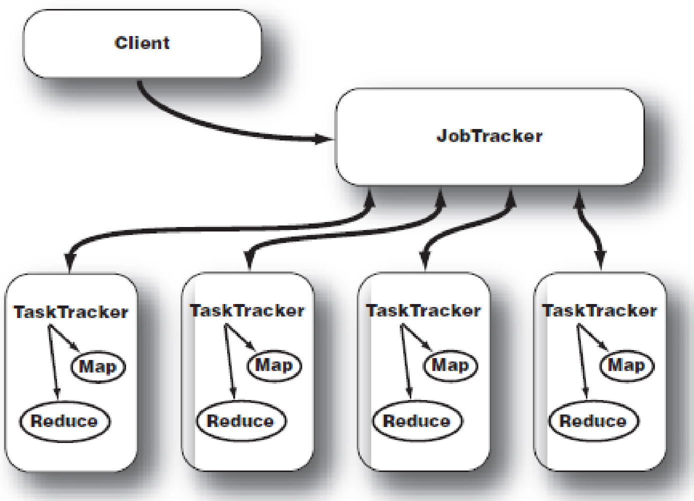


Figure JobTracker and TaskTracker interaction.

After a client calls the JobTracker to begin a data processing job, the JobTracker partitions the work and assigns different map and reduce tasks to each TaskTracker in the cluster.

The following figure shows topology of typical Hadoop cluster.

This topology features a master node running the NameNode and JobTracker daemons and a standalone node with the SNN in case the master node fails. For small clusters, the SNN can reside on one of the slave nodes. On the other hand, for large clusters, separate the NameNode and JobTracker on two machines. The slave machines each host a DataNode and TaskTracker, for running tasks on the same node where their data is stored.

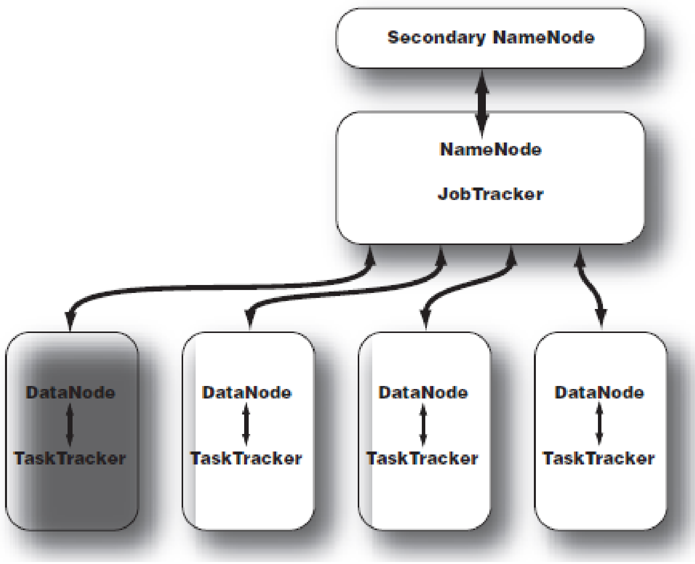


Figure Topology of a typical Hadoop cluster.

It's a master/slave architecture in which the NameNode and JobTracker are masters and the DataNodes and TaskTrackers are slaves.

<u>Introducing and Configuring Hadoop cluster (Local, Pseudo-distributed mode, Fully Distributed mode)</u>

<u>Local (standalone) mode</u>

The standalone mode is the default mode for Hadoop. With empty configuration files, Hadoop will run completely on the local machine. Because there's no need to communicate with other nodes, the standalone mode doesn't use HDFS, nor will it launch any of the Hadoop daemons. Its primary use is for developing and

debugging the application logic of a MapReduce program without the additional complexity of interacting with the daemons.
1. Place softwares into system downloads:
    i.hadoop-2.7.2.tar.gz
    ii.jdk-8u77-linux-i586.tar.gz
2. Extraction of tar files
Extract above two files in downloads itself. This can be done by right clicking on the tar file by selecting extract here option.
3. Rename the extracted file hadoop2.7.2 to hadoop.
4. Update ubuntu operating system
The ubuntu can be updated with the following command
user@user-ThinkCentre-E73:~$ sudo apt update
The above command asks the user enter password. Enter the corresponding password and press enter key.
5. Install openssh server
The next step is to install openssh server by typing the following command in terminal
user@user-ThinkCentre-E73:~$ sudo apt-get install openssh-server
6. Verify SSH installation
The first step is to check whether SSH is installed on your nodes. We can easily do this by use of the "which" UNIX command:
user@user-ThinkCentre-E73:~$ which ssh
7. Setup password less ssh to localhost
user@user-ThinkCentre-E73:~$ ssh-keygen -t rsa
8. Add the generated key into authorized keys.
user@user-ThinkCentre-E73**:~$cat /home/user/.ssh/id_rsa.pub>>/home/user/.ssh/authorized_keys**
9. To check local host connection type ssh localhost in terminal.
user@user-ThinkCentre-E73:~$ ssh localhost
10. After connecting to local host type exit.
user@user-ThinkCentre-E73:~$ exit
11. Open bashrc file and add the following lines at the end of file
user@user-ThinkCentre-E73:~$ sudo gedit .bashrc
```
export JAVA_HOME=/home/user/Downloads/jdk1.8.0_77
export PATH=$PATH:$JAVA_HOME/bin
HADOOP_HOME=/home/user/Downloads/hadoop
export PATH=$PATH:/home/user/Downloads/hadoop/bin
export PATH=$PATH:/home/user/Downloads/hadoop/sbin
```
12. Now apply all the changes into the current running system by typing the following command in terminal
user@user-ThinkCentre-E73:~$ source ~/.bashrc
14. Now verify the java version by giving the following command in terminal.
user@user-ThinkCentre-E73:~$ echo $JAVA_HOME
15. Now verify the hadooop version by giving the following command in terminal.
user@user-ThinkCentre-E73:~$ hadoop version

Pseudo-distributed mode
The pseudo-distributed mode is running Hadoop in a “cluster of one” with all daemons running on a single machine. This mode complements the standalone mode for debugging your code, allowing you to examine memory usage, HDFS input/output issues, and other daemon interactions. In order to work with pseudo distributed mode in addition to the above process the four XML files (core-site.xml, hdfs-site.xml, mapred-site.xml and yarn-site.xml) are to be updated as below.
core-site.xml
```
<configuration>
```

```
<property>
<name>fs.default.name</name>
<value>hdfs://localhost:9000</value>
</property>
<property>
<name>dfs.permissions</name>
<value>false</value>
</property>
</configuration>
```

hdfs-site.xml

```
<configuration>
<property>
<name>dfs.replication</name>
<value>1</value>
</property>
</configuration>
```

mapred-site.xml

```
<configuration>
<property>
<name>mapred.job.tracker</name>
<value>localhost:9001</value>
</property>
<property>
<name>mapreduce.framework.name</name>
<value>yarn</value>
</property>
</configuration>
```

yarn-site.xml

```
<configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
</configuration>
```

In core-site.xml and mapred-site.xml we specify the hostname and port of the NameNode and the JobTracker, respectively. In hdfs-site.xml we specify the default replication factor for HDFS, which should only be one because we're running on only one node. The yarn-site.xml is used to configure yarn into Hadoop.
The hadoop-env.sh file contains other variables for defining your Hadoop environment.To set environment variables we have to add following lines at the end of file in hadoop-env.sh.

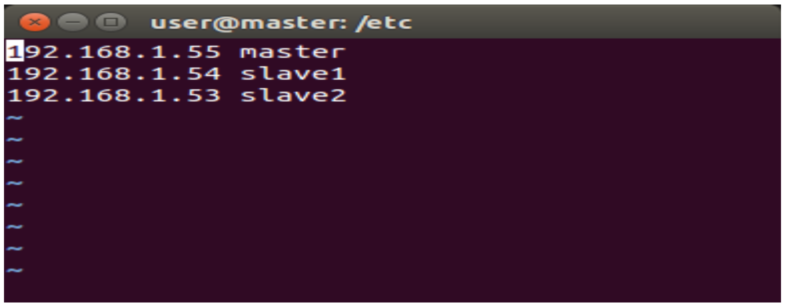
```
export JAVA_HOME=/home/user/Downloads/jdk1.8.0_77
Export HADOOP_HOME=/home/user/Downloads/hadoop
```

Fully distributed mode

The following are the three server names used to set up a full cluster.
master—The master node of the cluster and host of the NameNode and JobTracker daemons
master—The server that hosts the Secondary NameNode daemon
slave1, slave2, slave3, ...—The slave boxes of the cluster running both DataNode and TaskTracker daemons.

1. First decide how may nodes are to be grouped . Assume one of it as master and other as slave1, slave2 respectively. You can download hadoop-2.7.2.tar.gz file for hadoop
2. Extract it to a Downloads folder say, /home/user/Downloads on master.

NOTE: Master and all the slaves must have the same user name. Slaves should not have hadoop folder since it will be automatically created during installation.
3. To change name of host
$sudo gedit /etc/hostname
In terminal it will ask enter password **user123**
Now host name file is opened
Remove old name and type new name as **master, slave1 and slave2** respectively. Then save and exit. To apply these changes restart the master and slave nodes.
4. Add the association between the hostnames and the ip address for the master and the slaves on all the nodes in the /etc/hosts file. Make sure that the all the nodes in the cluster are able to ping to each other.
user@master:~$ cd /etc
user@master:/etc$ sudo vi hosts (OR) user@master:/etc$ sudo gedit hosts



```
user@master: /etc
192.168.1.55 master
192.168.1.54 slave1
192.168.1.53 slave2
~
~
~
~
~
~
~
~
```

In the slave1 and slave2 also perform the same operation.
5. Next we need to copy the public key to every slave node as well as the master node. Make sure that the master is able to do a password-less ssh to all the slaves.

```
$ ssh-keygen -t rsa
$ ssh-copy-id -i ~/.ssh/id_rsa.pub user@master
$ ssh-copy-id -i ~/.ssh/id_rsa.pub user@slave1
$ ssh-copy-id -i ~/.ssh/id_rsa.pub user@slave2
$ chmod 0600 ~/.ssh/authorized_keys
$ exit
```

6. open bashrc file and add the following lines at the end of file
user@user-ThinkCentre-E73:~$ sudo gedit .bashrc

```
export JAVA_HOME=/home/user/Downloads/jdk1.8.0_77
export PATH=$PATH:$JAVA_HOME/bin
export HADOOP_HOME=/home/user/Downloads/hadoop
export PATH=$PATH:$HADOOP_HOME/bin
export PATH=$PATH:$HADOOP_HOME/sbin
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
```

7. Edit Hadoop environment files
→ **Add following lines at the end of script in etc/hadoop/hadoop-env.sh**

```
export JAVA_HOME=/home/user/Downloads/jdk1.8.0_77
export HADOOP_HOME=/home/user/Downloads/hadoop
export HADOOP_OPTS=-Djava.net.preferIPv4Stack=true
```

→ **Add following lines at start of script in etc/hadoop/yarn-env.sh**

```
export JAVA_HOME=/home/user/Downloads/jdk1.8.0_77
export HADOOP_HOME=/home/user/Downloads/hadoop
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
```

8. Set the configuration files for fully distributed mode as below.

core-site.xml

```
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://master:9000</value>
  </property>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>/home/user/Downloads/hadoop/tmp</value>
  </property>
</configuration>
```

hdfs-site.xml :

```
<configuration>
   <property>
     <name>dfs.replication</name>
     <value>3</value>
   </property>
   <property>
     <name>dfs.permissions</name>
     <value>false</value>
   </property>
 </configuration>
```

mapred-site.xml

Add following lines mapred-site.xml :

```
<configuration>
 <property>
   <name>mapreduce.framework.name</name>
   <value>yarn</value>
 </property>
</configuration>
```

yarn-site.xml :

```
<configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
    <value>org.apache.hadoop.mapred.ShuffleHandler</value>
  </property>
  <property>
    <name>yarn.resourcemanager.resource-tracker.address</name>
    <value>master:8025</value>
  </property>
  <property>
    <name>yarn.resourcemanager.scheduler.address</name>
    <value>master:8030</value>
  </property>
  <property>
    <name>yarn.resourcemanager.address</name>
    <value>master:8040</value>
  </property>
 </configuration>
```

9. Add the slave entries in slaves file on master machine:

```
slave1
slave2
```

10. Use the following commands to copy hadoop from master node to slave nodes.

```
user@master:~/Downloads$ scp -r hadoop slave1:/home/user/Downloads/hadoop
user@master:~/Downloads$ scp -r hadoop slave2:/home/user/Downloads/hadoop
```

Running simple Example on standalone mode

Let's check a simple example of Hadoop. Hadoop installation delivers the following example MapReduce jar file, which provides basic functionality of MapReduce and can be used for calculating word counts in a given list of files.

Step 1. create input directory

You can create this input directory anywhere you would like to work and move to the created directory by using the following commands

```
user@user-ThinkCentre-E73:~$ mkdir input
user@user-ThinkCentre-E73:~$ cd input
```

step 2. Create some text files with some text as below

```
user@user-ThinkCentre-E73:~/input$ cat > f1.txt
```

Type some text and press CTRL+D to exit.

```
user@user-ThinkCentre-E73:~/input$ cat >f2.txt
```

Type some text and press CTRL+D to exit.

Step 3. come out of that directory by typing cd..

```
user@user-ThinkCentre-E73:~$ cd..
```

Step 4. Let's start the Hadoop process to count the total number of words in all the files available in the input directory, as follows:

```
user@user-ThinkCentre-E73:~$hadoop  jar  /home/user/Downloads/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar wordcount input outputt
```

Step 5. Now check output in terminal as

```
user@user-ThinkCentre-E73:~$ cat outputt/*
```

Running simple Example on pseudo distributed mode and fully distributed mode

1. Create a new directory on HDFS
   ```
   $ hdfs dfs -mkdir /in
   ```
2. Now copy the files from local directory to hdfs directory.
   ```
   $hdfs dfs -put /home/user/in/file /in/file
   ```
3. To excecute a program.
   ```
   $  hadoop jar /home/user/Downloads/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar wordcount /in /out
   ```
4. To get the output file from hdfs directory to local file system.
   ```
   $ hadoop fs -get /out /home/user/out
   ```
5. To see the output use the following command
   ```
   $ cat out/*
   ```