DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING – CYBER SECURITY

INDUSTRIAL ORIENTED MINI PROJECT/ INTERNSHIP/ SKILL DEVELOPMENT COURSE (BIG DATA-SPARK)

B.Tech. III Year II Sem (R22)

LIST OF EXPERIMENTS:

1. To Study of Big Data Analytics and Hadoop Architecture

   (i) know the concept of big data architecture (ii) know the concept of Hadoop architecture

2. Loading DataSet in to HDFS for Spark Analysis Installation of Hadoop and cluster management

   (i) Installing Hadoop single node cluster in ubuntu environment

   (ii) Knowing the differencing between single node clusters and multi-node clusters

   (iii) Accessing WEB-UI and the port number

   (iv) Installing and accessing the environments such as hive and sqoop

3. File management tasks & Basic linux commands

   (i) Creating a directory in HDFS

   (ii) Moving forth and back to directories

   (iii) Listing directory contents

   (iv) Uploading and downloading a file in HDFS

   (v) Checking the contents of the file

   (vi) Copying and moving files

   (vii) Copying and moving files between local to HDFS environment

   (viii) Removing files and paths (ix) Displaying few lines of a file

   (x) Display the aggregate length of a file (xi) Checking the permissions of a file

   (xii) Zipping and unzipping the files with & without permission pasting it to a location

   (xiii) Copy, Paste commands

4. Map-reducing

   (i) Definition of Map-reduce

   (ii) Its stages and terminologies

   (iii) Word-count program to understand map-reduce (Mapper phase, Reducer phase, Driver code)

5. Implementing Matrix-Multiplication with Hadoop Map-reduce

6. Compute Average Salary and Total Salary by Gender for an Enterprise.

7. (i) Creating hive tables (External and internal)

(ii) Loading data to external hive tables from sql tables(or)Structured c.s.v using scoop

(iii) Performing operations like filterations and updations

(iv) Performing Join (inner, outer etc)

(v) Writing User defined function on hive tables

8. Create a sql table of employees Employee table with id,designation Salary table (salary ,dept id) Create external table in hive with similar schema of above tables,Move data to hive using scoop and load the contents into tables,filter a new table and

write a UDF to encrypt the table with AES-algorithm, Decrypt it with key to show contents

9. (i) Pyspark Definition(Apache Pyspark) and difference between Pyspark, Scala, pandas

(ii) Pyspark files and class methods

(iii) get(file name)

(iv) get root directory()

10. Pyspark -RDD'S

(i) what is RDD's?

(ii) ways to Create RDD

(iii) parallelized collections

(iv) external dataset

(v) existing RDD's

(vi) Spark RDD's operations (Count, foreach(), Collect, join,Cache()

11. Perform pyspark transformations

(i) map and flatMap

(ii) to remove the words, which are not necessary to analyze this text.

(iii) groupBy

(iv) What if we want to calculate how many times each word is coming in corpus ?

(v) How do I perform a task (say count the words 'spark' and 'apache' in rdd3) separatly on each partition and get the output of the task performed in these partition ?

(vi) unions of RDD (vii) join two pairs of RDD Based upon their key

12. Pyspark sparkconf-Attributes and applications (i) What is Pyspark spark conf () (ii) Using spark conf create a spark session to write a dataframe to read details in a c.s.v and later move that c.s.v to another location

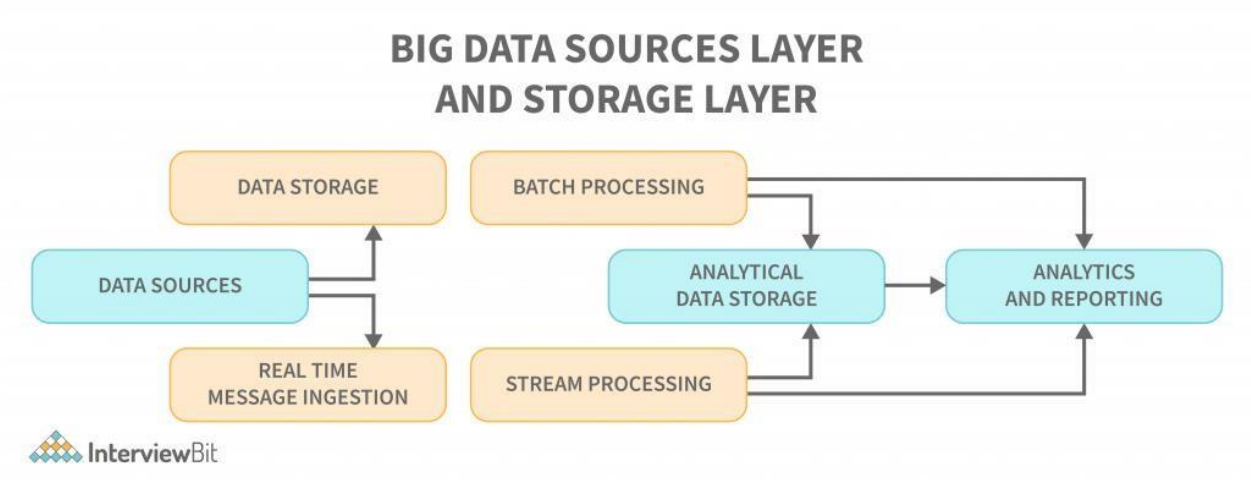
MODULE-1

TO STUDY OF BIG DATA ANALYTICS AND HADOOP ARCHITECTURE

(I) KNOW THE CONCEPT OF BIG DATA ARCHITECTURE (II) KNOW THE CONCEPT OF HADOOP ARCHITECTURE

(I) KNOW THE CONCEPT OF BIG DATA ARCHITECTURE

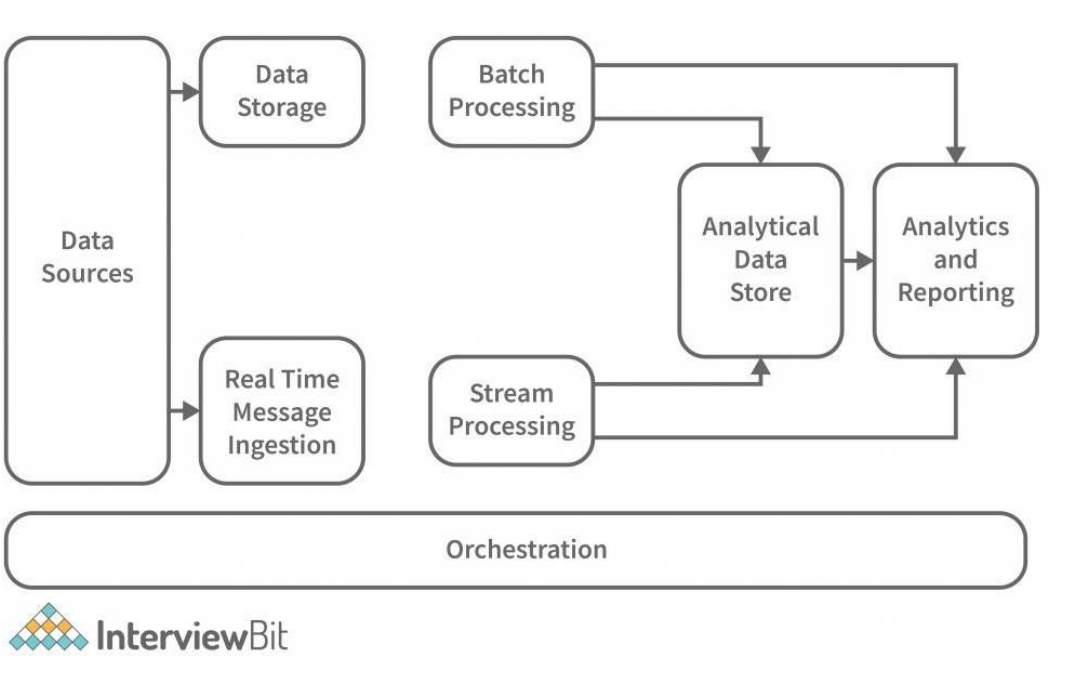
Big data architecture is a comprehensive solution to deal with an enormous amount of data. It details the blueprint for providing solutions and infrastructure for dealing with big data based on a company's demands. It clearly defines the components, layers, and methods of communication. The reference point is the ingestion, processing, storing, managing, accessing, and analysing of the data. A big data architecture typically looks like the one shown below, with the following layers:



- A database management system that handles data ingestion, processing, and analysis is too complex or too large to handle a traditional architecture. A traditional architecture handles all of that in one fell swoop, while a database management system handles it in chunks.
- Some organisations have threshold values for gigabytes or terabytes, while others, even millions of gigabytes or terabytes, are not good enough.
- As an example, if you look at storage systems and commodity markets, the values and costs of storage have significantly decreased due to this occurrence. There is lots of data that requires different methods to be stored.
- A big data architecture addresses some of these problems by providing a scalable and efficient method of storage and processing data. Some of them are batch-related data that occurs at a particular time and therefore the jobs must be scheduled in the same way as batch data. Streaming class jobs require a real-time streaming pipeline to be built to meet all of their demands. This process is accomplished through big data architecture.

There is more than one workload type involved in big data systems, and they are broadly classified as follows:

1. Merely batching data where big data-based sources are at rest is a data processing situation.
2. Real-time processing of big data is achievable with motion-based processing.
3. The exploration of new interactive big data technologies and tools.
4. The use of machine learning and predictive analysis.



- **Data Sources:** All of the sources that feed into the data extraction pipeline are subject to this definition, so this is where the starting point for the big data pipeline is located. Data sources, open and third-party, play a significant role in architecture. Relational databases, data warehouses, cloud-based data warehouses, SaaS applications, real-time data from company servers and sensors such as IoT devices, third-party data providers, and also static files such as Windows logs, comprise several data sources. Both batch processing and real-time processing are possible. The data managed can be both batch processing and real-time processing.

- **Data Storage:** There is data stored in file stores that are distributed in nature and that can hold a variety of format-based big files. It is also possible to store large numbers of different format-based big files in the data lake. This consists of the data that is managed for batch built operations and is saved in the file stores. We provide HDFS, Microsoft Azure, AWS, and GCP storage, among other blob containers.

- **Batch Processing:** Each chunk of data is split into different categories using long-running jobs, which filter and aggregate and also prepare data for analysis. These jobs typically require sources, process them, and deliver the processed files to new files. Multiple approaches to batch processing are employed, including Hive jobs, U-SQL jobs, Sqoop or Pig and custom map reducer jobs written in any one of the Java or Scala or other languages such as Python.

- **Real Time-Based Message Ingestion:** A real-time streaming system that caters to the data being generated in a sequential and uniform fashion is a batch processing system. When compared to batch processing, this includes all real-time streaming systems that cater to the data being generated at the time it is received. This data mart or store, which receives all incoming messages and discards them into a folder for data processing, is usually the only one that needs to be contacted. Message-based ingestion stores such as Apache Kafka, Apache Flume, Event hubs from Azure, and others, on the other hand, must be used if message-based processing is required. The delivery process, along with other message queuing semantics, is generally more reliable.

- **Stream Processing:** Real-time message ingest and stream processing are different. The latter uses the ingested data as a publish-subscribe tool, whereas the former takes into account all of the ingested data in the first place and then utilises it as a publish-subscribe tool. Stream processing, on the other hand, handles all of that streaming data in the form of windows or streams and writes it to the sink. This includes Apache Spark, Flink, Storm, etc.

- **Analytics-Based Datastore:** In order to analyze and process already processed data, analytical tools use the data store that is based on HBase or any other NoSQL data warehouse technology. The data can be presented with the help of a hive database, which can provide metadata abstraction, or interactive use of a hive database, which can provide metadata abstraction in the data store. NoSQL databases like HBase or Spark SQL are also available.

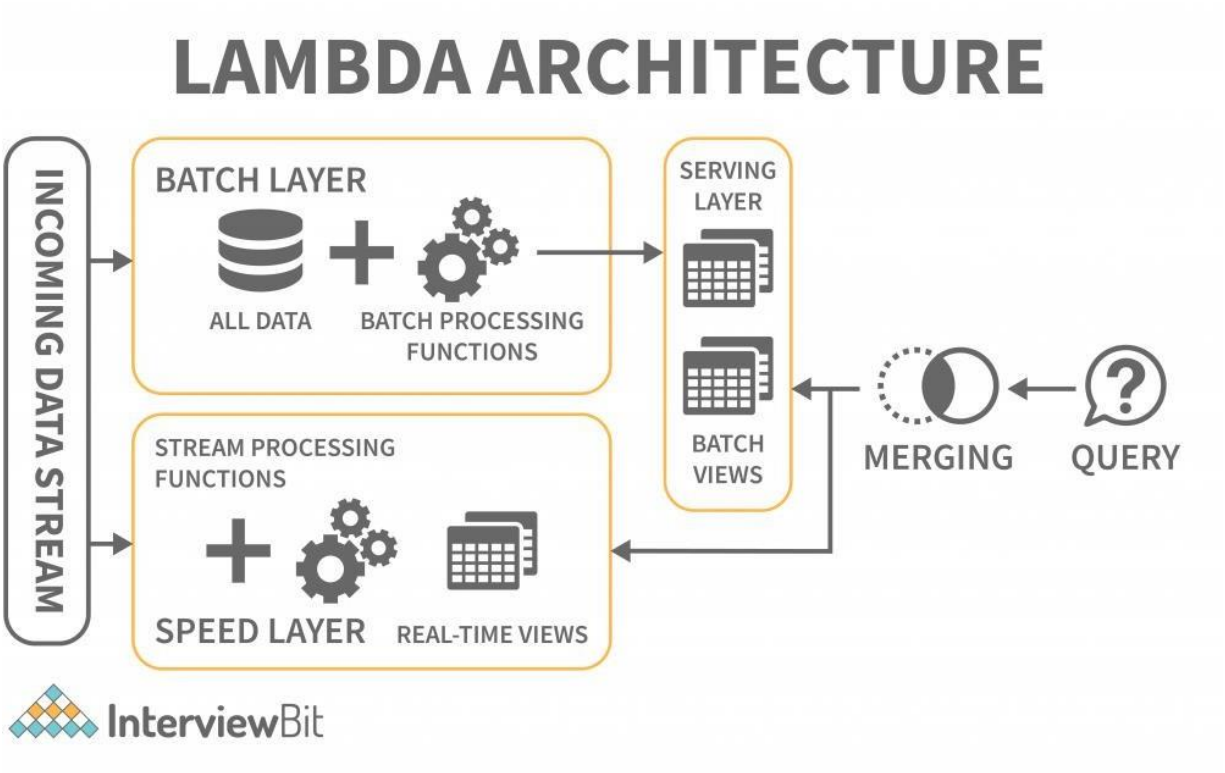
- **Reporting and Analysis:** The generated insights, on the other hand, must be processed and that is effectively accomplished by the reporting and analysis tools that utilize embedded technology and a solution to produce useful graphs, analysis, and insights that are beneficial to the businesses. For example, Cognos, Hyperion, and others.

- **Orchestration:** Data-based solutions that utilise big data are data-related tasks that are repetitive in nature, and which are also contained in workflow chains that can transform the source data and also move data across sources as well as sinks and loads in stores. Sqoop, oozie, data factory, and others are just a few examples.

TYPES OF BIG DATA ARCHITECTURE

Lambda Architecture

A single Lambda architecture handles both batch (static) data and real-time processing data. It is employed to solve the problem of computing arbitrary functions. In this deployment model, latency is reduced and negligible errors are preserved while retaining accuracy. The big data architecture illustrated below is similar to that described:



The lambda architecture is comprised of these layers:

- **Batch Layer:** The batch layer of the lambda architecture saves incoming data in its entirety as batch views. The batch views are used to prepare the indexes. The data is immutable, and only copies of the original data are created and preserved. The batch layer ensures consistency by making the data append-only. It is the first layer in the lambda architecture that saves incoming data in its entirety as batch views. The data cannot be changed, and only copies of the original data are created and preserved. The data that is saved is immutable, meaning that it cannot be changed, and only copies of the original data are preserved and stored. The data that is saved is append-only, which ensures that it is prepared before it is presented. The master dataset and then pre-computing the batch views are handled this way.

- **Speed Layer:** The speed layer delivers data straight to the batch layer, which is responsible for computing incremental data. However, the speed layer itself may also be reduced in latency by reducing the number of computations. The stream layer processes the processed data from the speed layer to produce error correction.

- **Serving Layer:** The batch views and the speed outcomes traverse to the serving layer as a result of the batch layers batch views. The serving layer indexes the views and parallelizes them to ensure users' queries are fast and are exempt from delays.

KAPPA ARCHITECTURE

When compared to Lambda architecture, Kappa architecture is also intended to handle both real-time streaming and batch processing data. The Kappa architecture, in addition to reducing the additional cost that comes from the Lambda architecture, replaces the data sourcing medium with message queues. The messaging engines store a sequence of data in the analytical databases, which are then read and converted into appropriate format before being saved for the end-user.

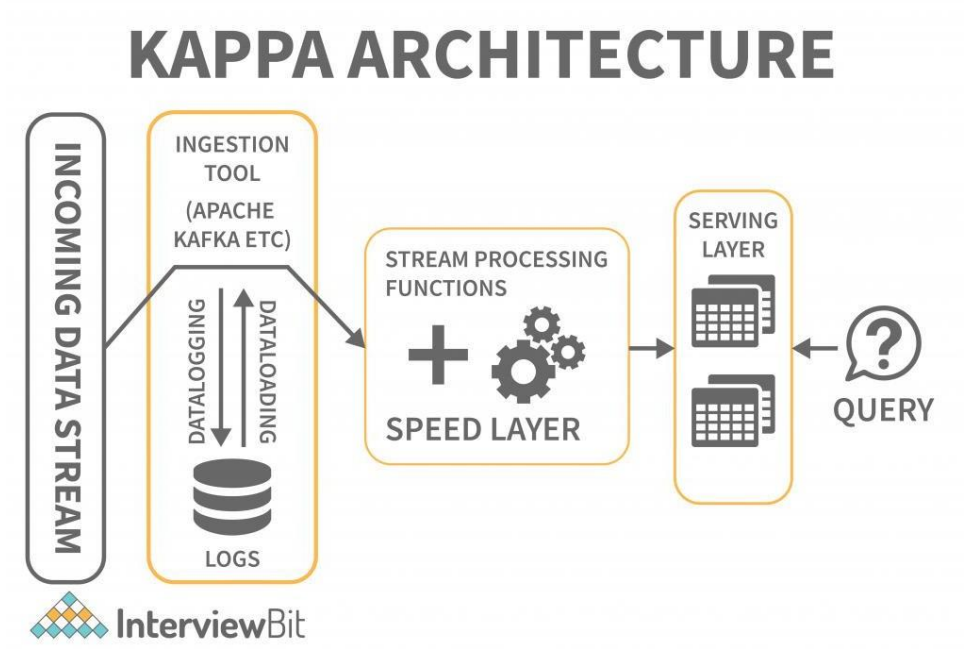
The architecture makes it easy to access real-time information by reading and transforming the message data into a format that is easily accessible to end users. It also provides additional outputs by allowing previously saved data to be taken into account.

The batch layer was eliminated in the Kappa architecture, and the speed layer was enhanced to provide reprogramming capabilities. The key difference with the Kappa architecture is that all the data is presented as a series or stream. Data transformation is achieved through the steam engine, which is the central engine for data processing.

Big Data Tools and Techniques

A big data tool can be classified into the four buckets listed below based on its practicability.

1. Massively Parallel Processing (MPP)
2. No-SQL Databases
3. Distributed Storage and Processing Tools
4. Cloud Computing Tools



Benefits of Big Data Architecture

- **High-performance parallel computing:** Big data architectures employ parallel computing, in which multiprocessor servers perform lots of calculations at the same time to speed up the process. Large data sets can be processed quickly by parallelising them on multiprocessor servers. Part of the job can be handled simultaneously.

- **Elastic scalability:** Big Data architectures can be scaled horizontally, allowing the environment to be tuned to the size of the workloads. A big data solution is usually operated in the cloud, where you only have to pay for the storage and processing resources you actually utilise.

- **Freedom of choice:** Big Data architectures may use various platforms and solutions in the marketplace, such as Azure-managed services, MongoDB Atlas, and Apache technologies. You can pick the right combination of solutions for your specific workloads, existing systems, and IT expertise levels to achieve the best result.

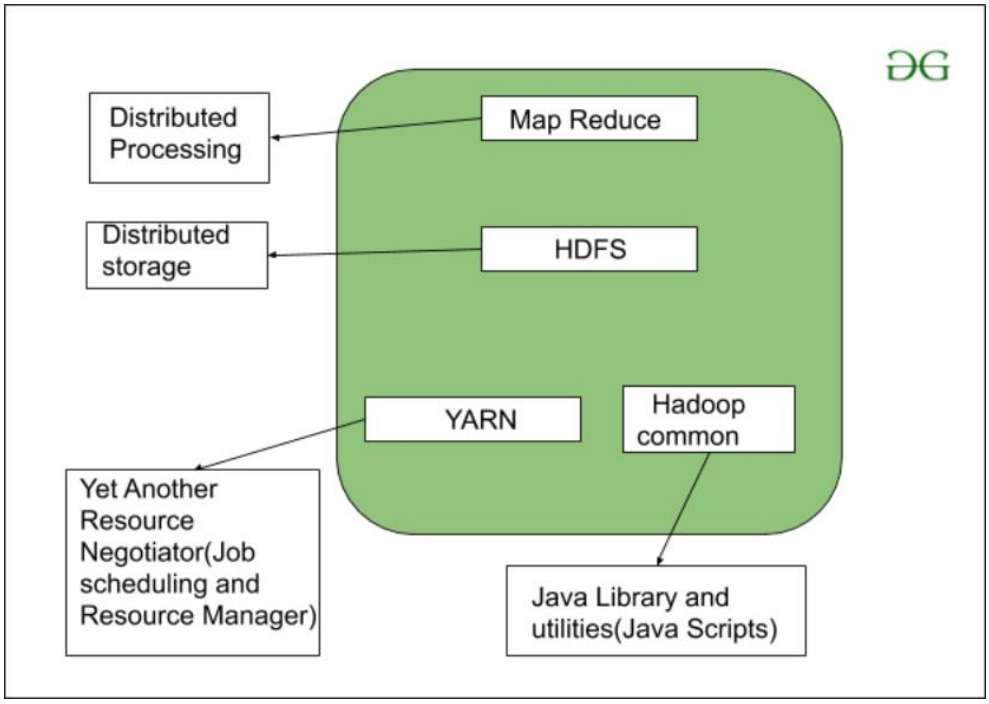
- **The ability to interoperate with other systems:** You can use Big Data architecture components for IoT processing and BI as well as analytics workflows to create integrated platforms across different types of workloads.

(II) KNOW THE CONCEPT OF HADOOP ARCHITECTURE

Hadoop – Architecture

As we all know Hadoop is a framework written in Java that utilizes a large cluster of commodity hardware to maintain and store big size data. Hadoop works on MapReduce Programming Algorithm that was introduced by Google. Today lots of Big Brand Companies are using Hadoop in their Organization to deal with big data, eg. Facebook, Yahoo, Netflix, eBay, etc. The Hadoop Architecture Mainly consists of 4 components.

- MapReduce
- HDFS(Hadoop Distributed File System)
- YARN(Yet Another Resource Negotiator)
- Common Utilities or Hadoop Common

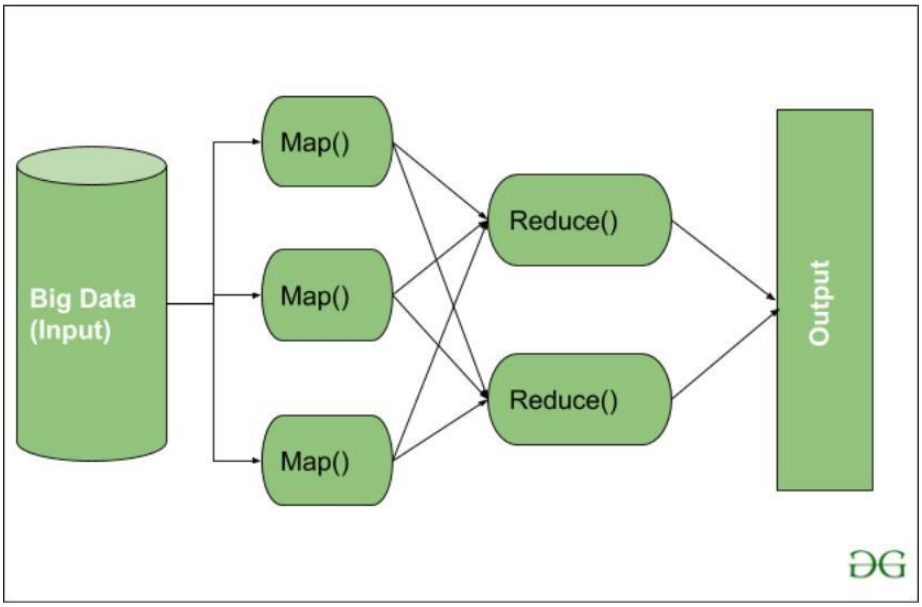


Let's understand the role of each one of this component in detail.

1. MapReduce

MapReduce nothing but just like an Algorithm or a data structure that is based on the YARN framework. The major feature of MapReduce is to perform the distributed processing in parallel in a Hadoop cluster which Makes Hadoop working so fast. When you are dealing with Big Data, serial processing is no more of any use. MapReduce has mainly 2 tasks which are divided phase-wise:

In first phase, **Map** is utilized and in next phase **Reduce** is utilized.



Here, we can see that the *Input* is provided to the Map() function then it's *output* is used as an input to the Reduce function and after that, we receive our final output. Let's understand What this Map() and Reduce() does.

As we can see that an Input is provided to the Map(), now as we are using Big Data. The Input is a set of Data. The Map() function here breaks this Data Blocks into **Tuples** that are nothing but a key-value pair. These key-value pairs are now sent as input to the Reduce(). The Reduce() function then combines this broken Tuples or key-value pair based on its Key value and form set of Tuples, and perform some operation like sorting, summation type job, etc. which is then sent to the final Output Node. Finally, the Output is Obtained.

The data processing is always done in Reducer depending upon the business requirement of that industry. This is How First Map() and then Reduce is utilized one by one.
Let's understand the *Map Task* and *Reduce Task* in detail.
MapTask:

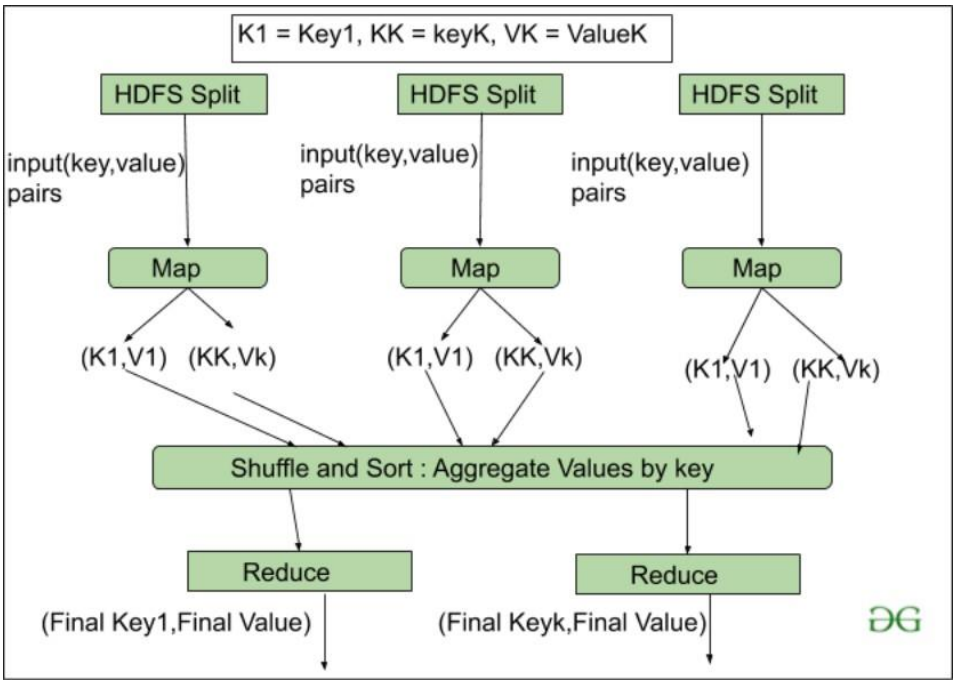
- **RecordReader** The purpose of *recordreader* is to break the records. It is responsible for providing key-value pairs in a Map() function. The key is actually is its locational information and value is the data associated with it.
- **Map:** A map is nothing but a user-defined function whose work is to process the Tuples obtained from record reader. The Map() function either does not generate any key-value pair or generate multiple pairs of these tuples.

- **Combiner:** Combiner is used for grouping the data in the Map workflow. It is similar to a Local reducer. The intermediate key-value that are generated in the Map is combined with the help of this combiner. Using a combiner is not necessary as it is optional.
- **Partitionar:** Partitional is responsible for fetching key-value pairs generated in the Mapper Phases. The partitioner generates the shards corresponding to each reducer. Hashcode of each key is also fetched by this partition. Then partitioner performs it's(Hashcode) modulus with the number of reducers(*key.hashcode()%(number of reducers)).*

ReduceTask

- **Shuffle and Sort:** The Task of Reducer starts with this step, the process in which the Mapper generates the intermediate key-value and transfers them to the Reducer task is known as *Shuffling*. Using the Shuffling process the system can sort the data using its key value.

  Once some of the Mapping tasks are done Shuffling begins that is why it is a faster process and does not wait for the completion of the task performed by Mapper.
- **Reduce:** The main function or task of the Reduce is to gather the Tuple generated from Map and then perform some sorting and aggregation sort of process on those key-value depending on its key element.
- **OutputFormat:** Once all the operations are performed, the key-value pairs are written into the file with the help of record writer, each record in a new line, and the key and value in a space-separated manner.



2. HDFS

HDFS(Hadoop Distributed File System) is utilized for storage permission. It is mainly designed for working on commodity Hardware devices(inexpensive devices), working on a distributed file system design. HDFS is designed in such a way that it believes more in storing the data in a large chunk of blocks rather than storing small data blocks.

HDFS in Hadoop provides Fault-tolerance and High availability to the storage layer and the other devices present in that Hadoop cluster. Data storage Nodes in HDFS.

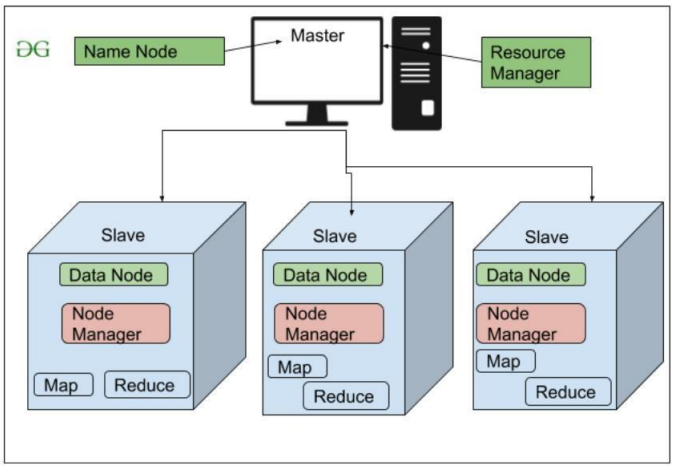
- NameNode(Master)
- DataNode(Slave)

  NameNode: NameNode works as a Master in a Hadoop cluster that guides the Datanode(Slaves). Namenode is mainly used for storing the Metadata i.e. the data about the data. Meta Data can be the transaction logs that keep track of the user's activity in a Hadoop cluster.

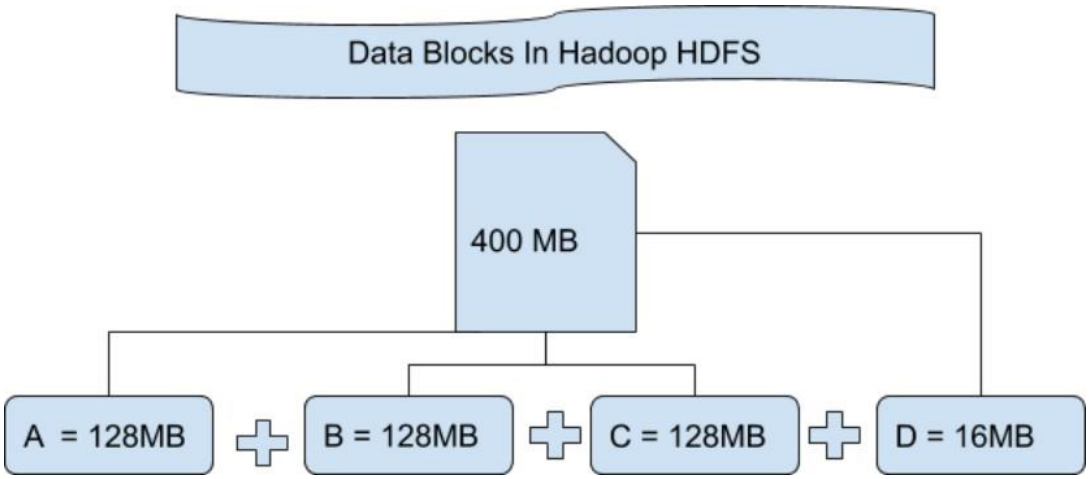
  Meta Data can also be the name of the file, size, and the information about the location(Block number, Block ids) of Datanode that Namenode stores to find the closest DataNode for Faster Communication. Namenode instructs the DataNodes with the operation like delete, create, Replicate, etc.

  DataNode: DataNodes works as a Slave DataNodes are mainly utilized for storing the data in a Hadoop cluster, the number of DataNodes can be from 1 to 500 or even more than that. The more number of DataNode, the Hadoop cluster will be able to store more data. So it is advised that the DataNode should have High storing capacity to store a large number of file blocks.

High Level Architecture Of Hadoop



File Block In HDFS: Data in HDFS is always stored in terms of blocks. So the single block of data is divided into multiple blocks of size 128MB which is default and you can also change it manually.



Let's understand this concept of breaking down of file in blocks with an example. Suppose you have uploaded a file of 400MB to your HDFS then what happens is this file got divided into blocks of 128MB+128MB+128MB+16MB = 400MB size. Means 4 blocks are created each of 128MB except the last one. Hadoop doesn't know or it doesn't care about what data is stored in these blocks so it considers the final file blocks as a partial record as it does not have any idea regarding it. In the Linux file system, the size of a file block is about 4KB which is very much less than the default size of file blocks in the Hadoop file system. As we all know Hadoop is mainly configured for storing the large size data which is in petabyte, this is what makes Hadoop file system different from other file systems as it can be scaled, nowadays file blocks of 128MB to 256MB are considered in Hadoop.

Replication In HDFS Replication ensures the availability of the data. Replication is making a copy of something and the number of times you make a copy of that particular thing can be expressed as it's Replication Factor. As we have seen in File blocks that the HDFS stores the data in the form of various blocks at the same time Hadoop is also configured to make a copy of those file blocks.

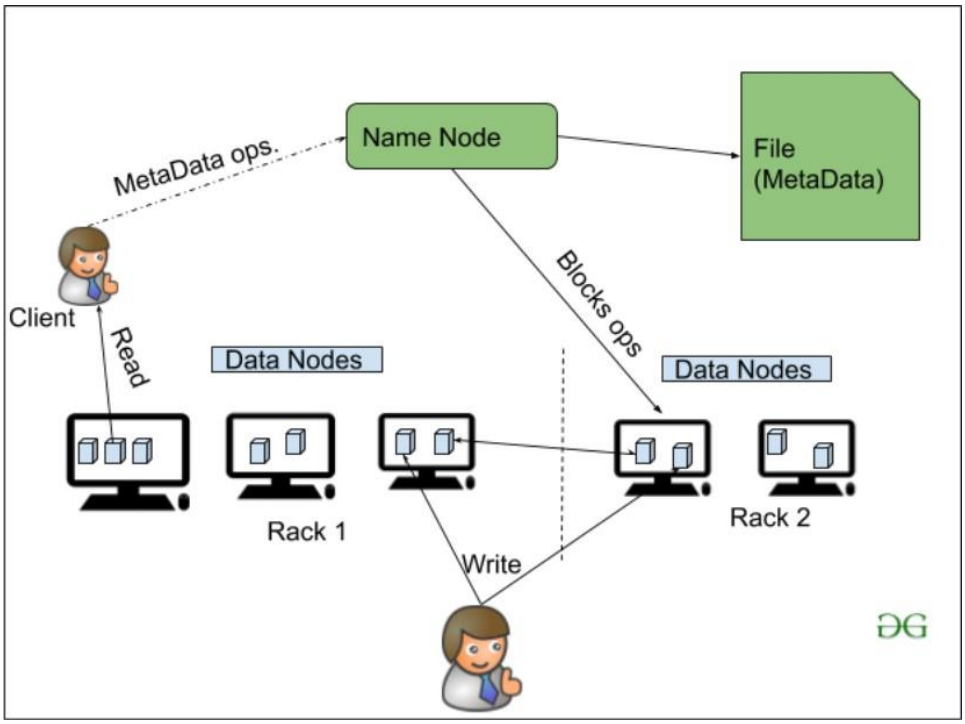
By default, the Replication Factor for Hadoop is set to 3 which can be configured means you can change it manually as per your requirement like in above example we have made 4 file blocks which means that 3 Replica or copy of each file block is made means total of 4×3 = 12 blocks are made for the backup purpose.

This is because for running Hadoop we are using commodity hardware (inexpensive system hardware) which can be crashed at any time. We are not using the supercomputer for our Hadoop setup. That is why we need such a feature in HDFS which can make copies of that file blocks for backup purposes, this is known as fault tolerance.

Now one thing we also need to notice that after making so many replica's of our file blocks we are wasting so much of our storage but for the big brand organization the data is very much important than the storage so nobody cares for this extra storage. You can configure the Replication factor in your *hdfs-site.xml* file.

Rack Awareness The rack is nothing but just the physical collection of nodes in our Hadoop cluster (maybe 30 to 40). A large Hadoop cluster is consists of so many Racks . with the help of this Racks information Namenode chooses the closest Datanode to achieve the maximum performance while performing the read/write information which reduces the Network Traffic.

HDFS Architecture



3. YARN(Yet Another Resource Negotiator)

YARN is a Framework on which MapReduce works. YARN performs 2 operations that are Job scheduling and Resource Management. The Purpose of Job schedular is to divide a big task into small jobs so that each job can be assigned to various slaves in a Hadoop cluster and Processing can be Maximized. Job Scheduler also keeps track of which job is important, which job has more priority, dependencies between the jobs and all the other information like job timing, etc. And the use of Resource Manager is to manage all the resources that are made available for running a Hadoop cluster.

Features of YARN

- Multi-Tenancy
- Scalability
- Cluster-Utilization
- Compatibility

4. Hadoop common or Common Utilities

Hadoop common or Common utilities are nothing but our java library and java files or we can say the java scripts that we need for all the other components present in a Hadoop cluster. these utilities are used by HDFS, YARN, and MapReduce for running the cluster. Hadoop Common verify that Hardware failure in a Hadoop cluster is common so it needs to be solved automatically in software by Hadoop Framework.

MODULE-2

LOADING DATASET IN TO HDFS FOR SPARK ANALYSIS INSTALLATION OF HADOOP AND CLUSTER MANAGEMENT

(i) Installing Hadoop single node cluster in ubuntu environment

(ii) Knowing the differencing between single node clusters and multi-node clusters

(iii) Accessing WEB-UI and the port number

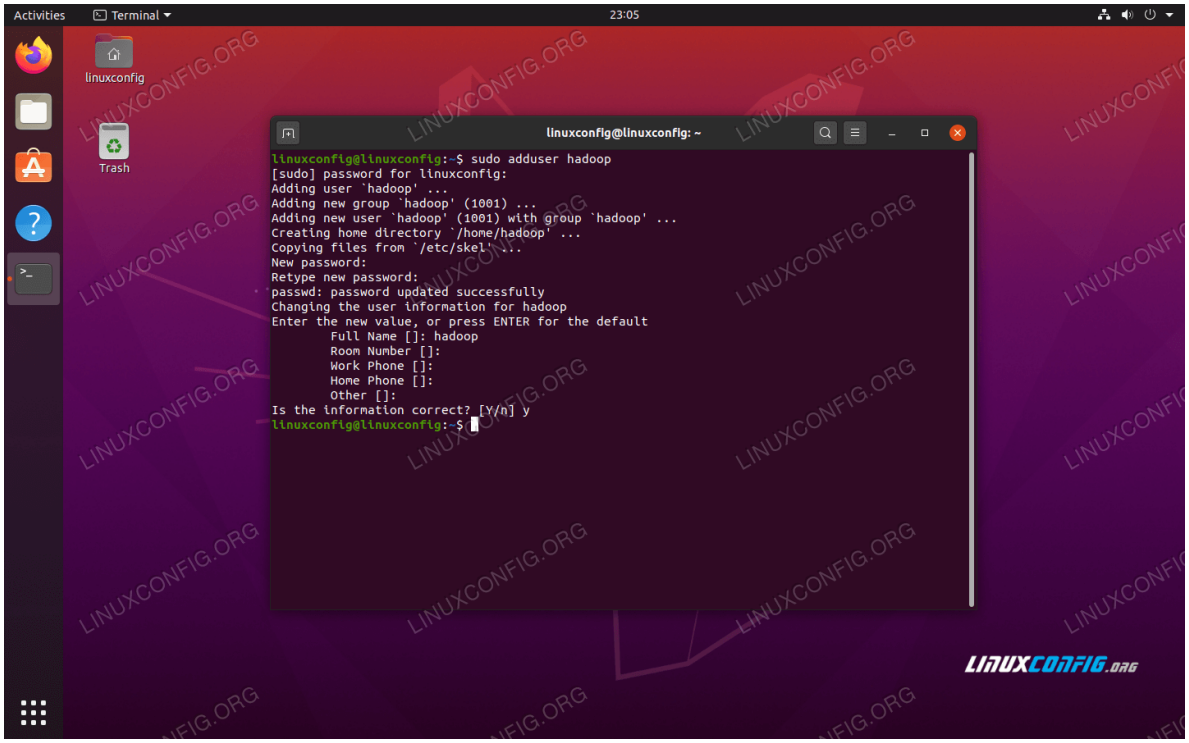
(iv) Installing and accessing the environments such as hive and sqoop

(I) INSTALLING HADOOP SINGLE NODE CLUSTER IN UBUNTU ENVIRONMENT

Create user for Hadoop environment

Hadoop should have its own dedicated user account on your system. To create one, open a terminal and type the following command. You'll also be prompted to create a password for the account.

```
$ sudo adduser hadoop
```



Install the Java prerequisite

Hadoop is based on Java, so you'll need to install it on your system before being able to use Hadoop. At the time of this writing, the current Hadoop version 3.1.3 requires Java 8, so that's what we will be installing on our system.

Use the following two commands to fetch the latest package lists in **apt** and install Java 8:

Configure passwordless SSH

Hadoop relies on SSH to access its nodes. It will connect to remote machines through SSH as well as your local machine if you have Hadoop running on it. So, even though we are only setting up Hadoop on our local machine in this tutorial, we still need to have SSH installed. We also have to configure passwordless SSH so that Hadoop can silently establish connections in the background.

STEP-1 We'll need both the OpenSSH Server and OpenSSH Client package. Install them with this command:

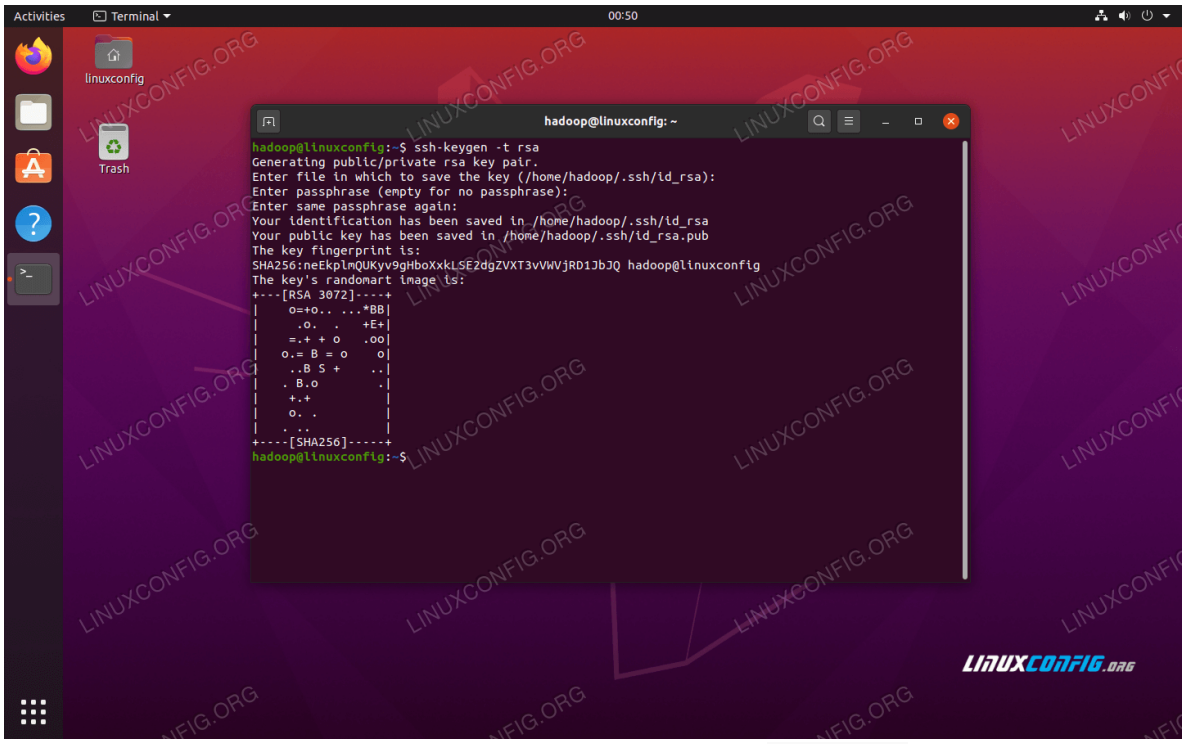
```
$ sudo apt install openssh-server openssh-client
```

STEP-2 Before continuing further, it's best to be logged into the **hadoop** user account we created earlier. To change users in your current terminal, use the following command:

```
$ su hadoop
```

STEP-3 With those packages installed, it's time to generate public and private key pairs with the following command. Note that the terminal will prompt you several times, but all you'll need to do is keep hitting **ENTER** to proceed.

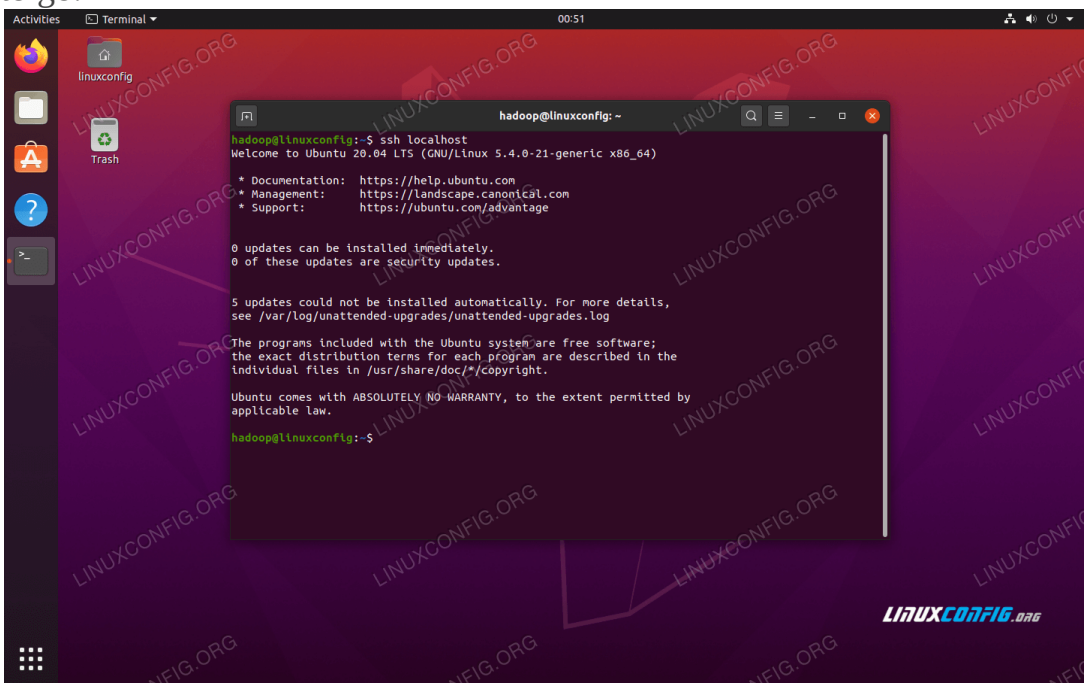
```
$ ssh-keygen -t rsa
```



STEP-4 Next, copy the newly generated RSA key in **id_rsa.pub** over to **authorized_keys**:

```
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

STEP-5 You can make sure that the configuration was successful by SSHing into localhost. If you are able to do it without being prompted for a password, you're good to go.



INSTALL HADOOP AND CONFIGURE RELATED XML FILES

Head over to Apache's website to download Hadoop. You may also use this command if you want to download the Hadoop version 3.1.3 binary directly:

```
$ wget https://downloads.apache.org/hadoop/common/hadoop-3.1.3/hadoop-3.1.3.tar.gz
```

Extract the download to the **hadoop** user's home directory with this command:

```
$ tar -xzvf hadoop-3.1.3.tar.gz -C /home/hadoop
```

SETTING UP THE ENVIRONMENT VARIABLE

The following **export** commands will configure the required Hadoop environment variables on our system. You can copy and paste all of these to your terminal (you may need to change line 1 if you have a different version of Hadoop):

```
export HADOOP_HOME=/home/hadoop/hadoop-3.1.3

export HADOOP_INSTALL=$HADOOP_HOME

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_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native

export PATH=$PATH:$HADOOP_HOME/sbin:$HADOOP_HOME/bin

export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib/native"
```

Source the **.bashrc** file in current login session:

```
$ source ~/.bashrc
```

Next, we will make some changes to the **hadoop-env.sh** file, which can be found in the Hadoop installation directory under **/etc/hadoop**. Use nano or your favorite text editor to open it:

```
$ nano ~/hadoop-3.1.3/etc/hadoop/hadoop-env.sh
```

Change the **JAVA_HOME** variable to where Java is installed. On our system (and probably yours too, if you are running Ubuntu 20.04 and have followed along with us so far), we change that line to:

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

That will be the only change we need to make in here. You can save your changes to the file and close it.

CONFIGURATION CHANGES IN CORE-SITE.XML FILE

The next change we need to make is inside the **core-site.xml** file. Open it with this command:

```
$ nano ~/hadoop-3.1.3/etc/hadoop/core-site.xml
```

```
fs.defaultFS
hdfs://localhost:9000


hadoop.tmp.dir
/home/hadoop/hadooptmpdata
```

Save your changes and close this file. Then, create the directory in which temporary data will be stored:

```
$ mkdir ~/hadooptmpdata
```

CONFIGURATION CHANGES IN HDFS-SITE.XML FILE

Create two new directories for Hadoop to store the Namenode and Datanode information.

```
$ mkdir -p ~/hdfs/namenode ~/hdfs/datanode
```

Then, edit the following file to tell Hadoop where to find those directories:

```
$ nano ~/hadoop-3.1.3/etc/hadoop/hdfs-site.xml
```

Make the following changes to the **hdfs-site.xml** file, before saving and closing it:

```
dfs.replication
1
dfs.name.dir
file:///home/hadoop/hdfs/namenode
dfs.data.dir
file:///home/hadoop/hdfs/datanode
```

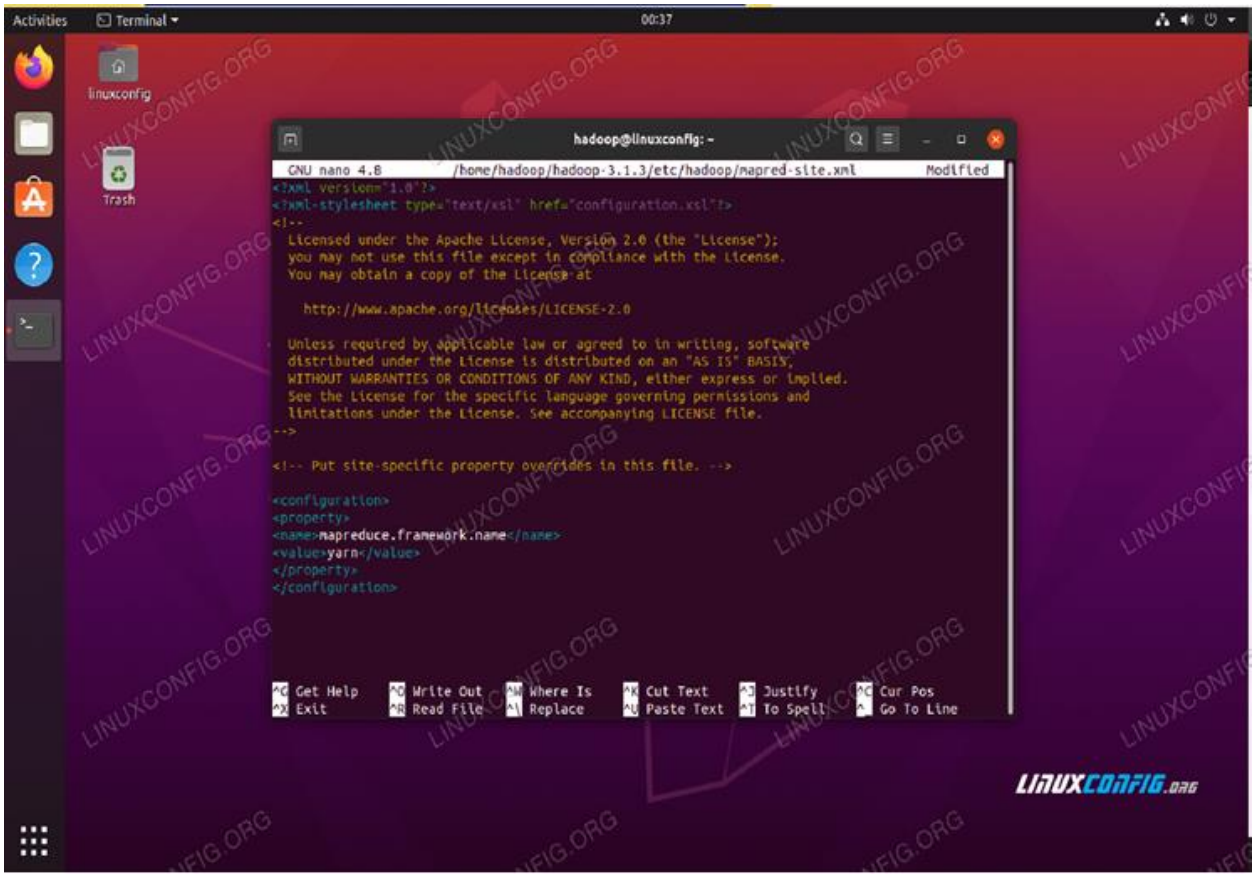
CONFIGURATION CHANGES IN MAPRED-SITE.XML FILE

Open the MapReduce XML configuration file with the following command:

```
$ nano ~/hadoop-3.1.3/etc/hadoop/mapred-site.xml
```

And make the following changes before saving and closing the file:

```
mapreduce.framework.name
yarn
```



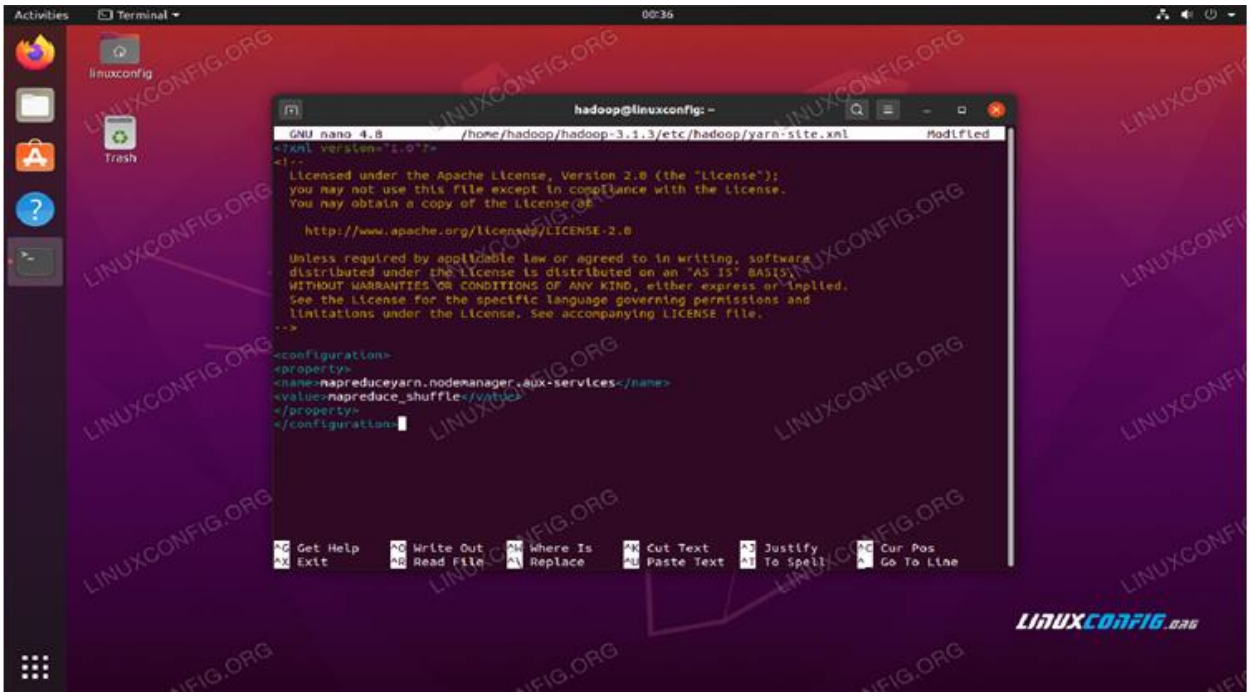
CONFIGURATION CHANGES IN YARN-SITE.XML FILE

Open the YARN configuration file with the following command

```
$ nano ~/hadoop-3.1.3/etc/hadoop/yarn-site.xml
```

Add the following entries in this file, before saving the changes and closing it:

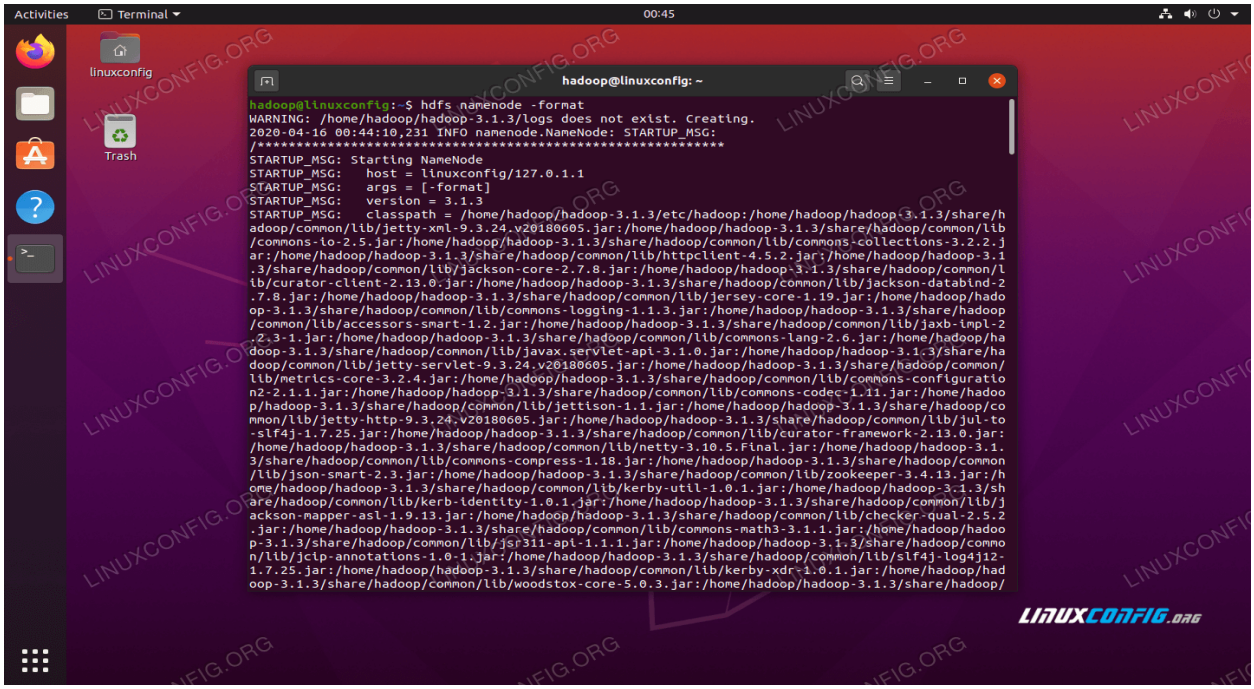
```
mapreduceyarn.nodemanager.aux-services
mapreduce_shuffle
```



STARTING THE HADOOP CLUSTER

Before using the cluster for the first time, we need to format the namenode. You can do that with the following command:

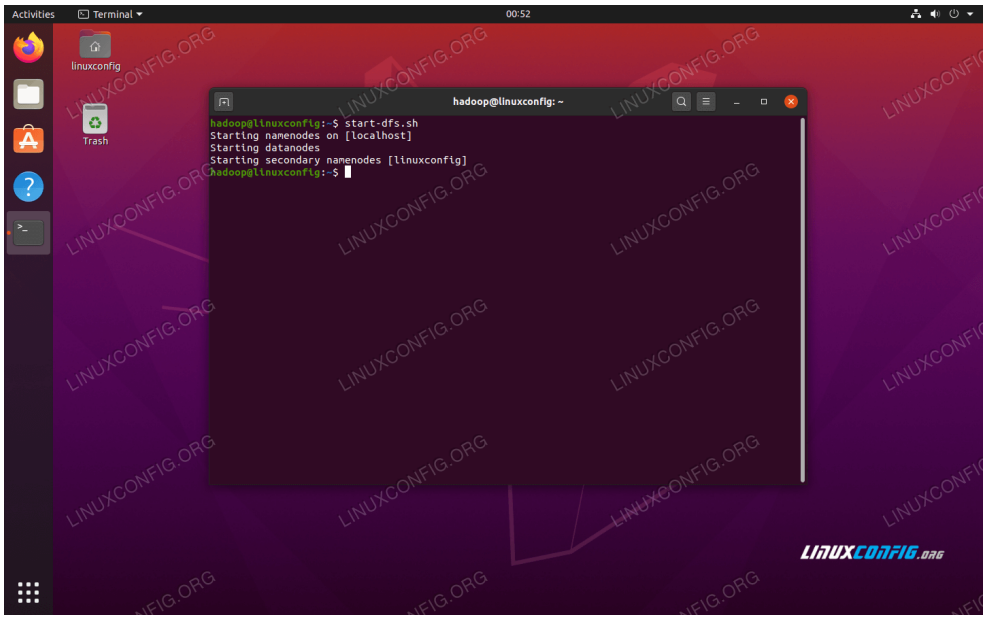
```
$ hdfs namenode -format
```



Your terminal will spit out a lot of information. As long as you don't see any error messages, you can assume it worked.

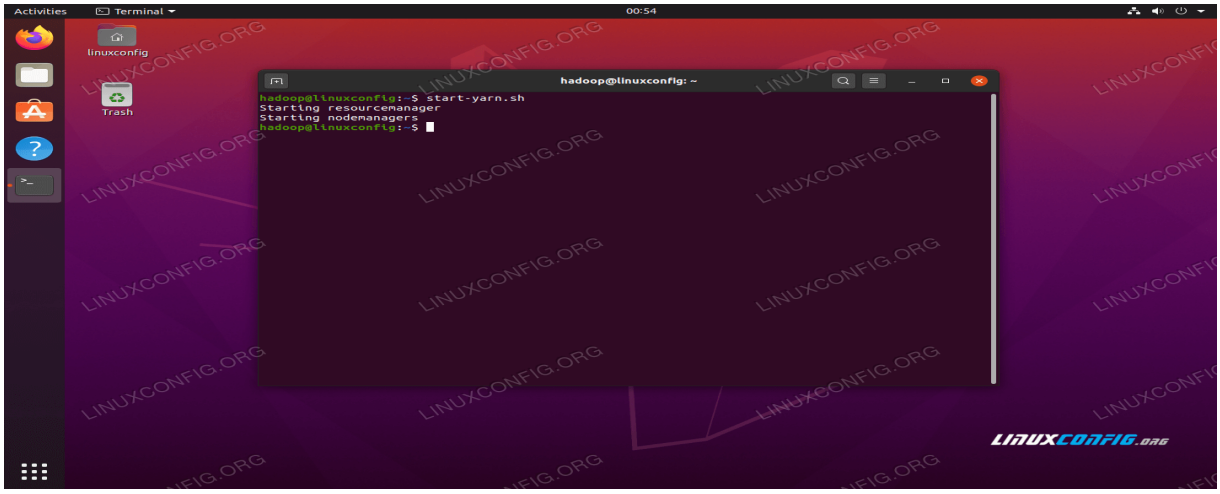
Next, start the HDFS by using the **start-dfs.sh** script:

```
$ start-dfs.sh
```



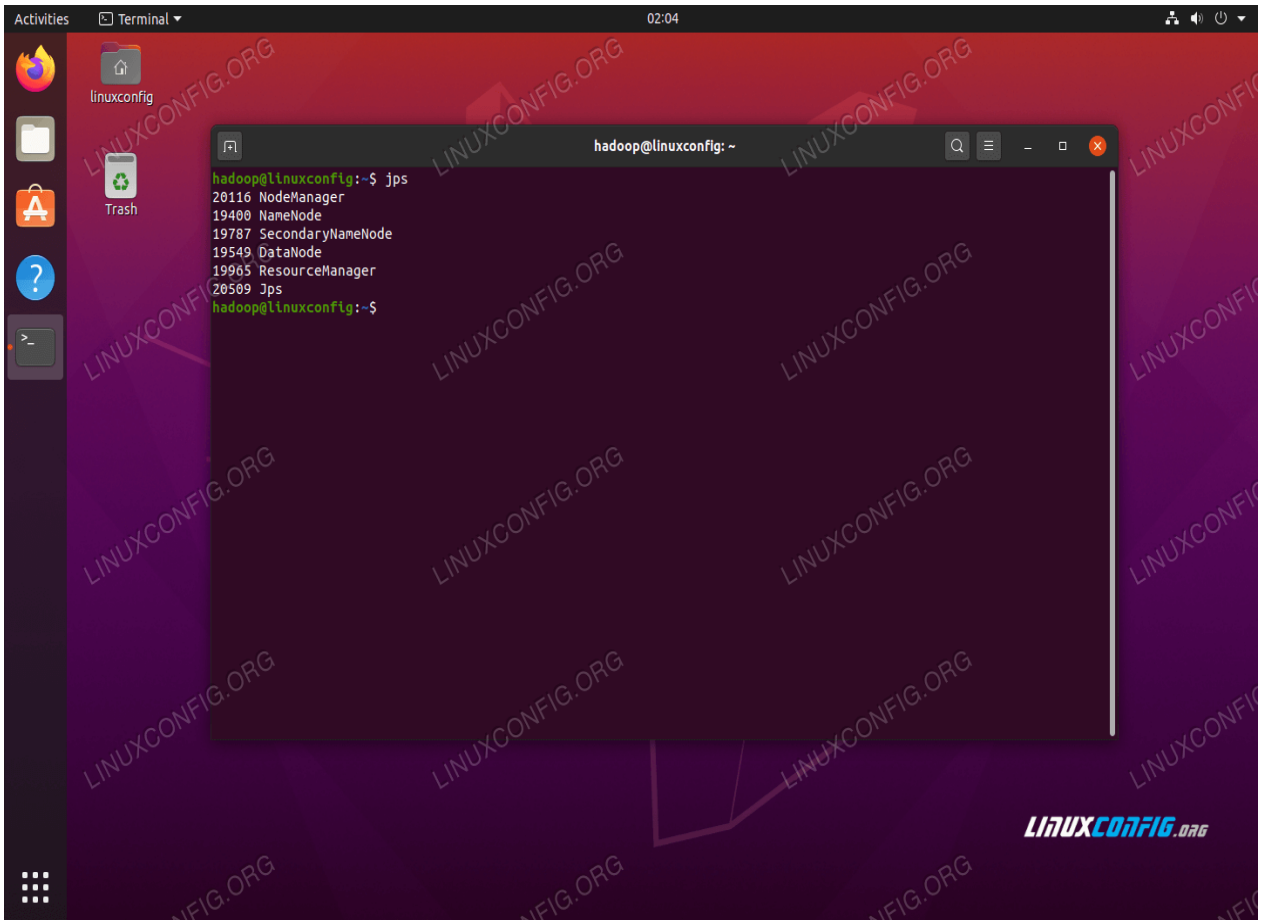
Now, start the YARN services via the `start-yarn.sh` script:

```
$ start-yarn.sh
```



To verify all the Hadoop services/daemons are started successfully you can use the **jps** command. This will show all the processes currently using Java that are running on your system.

```
$ jps
```

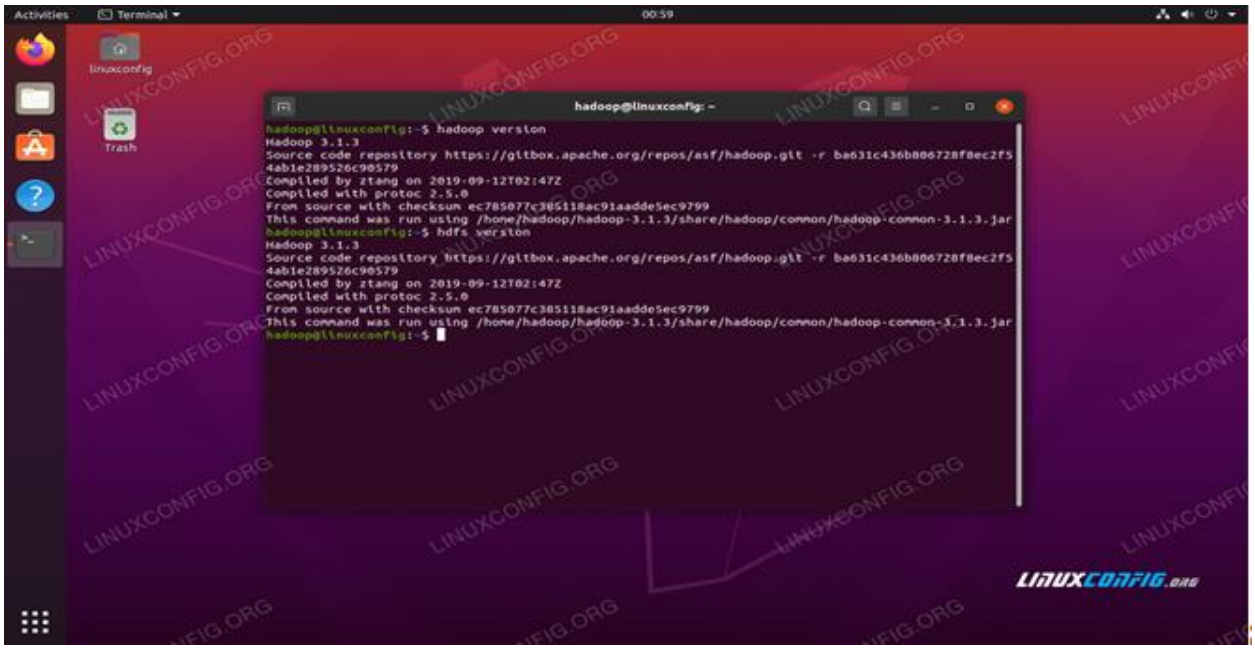


Now we can check the current Hadoop version with either of the following commands:

```
$ hadoop version
```

or

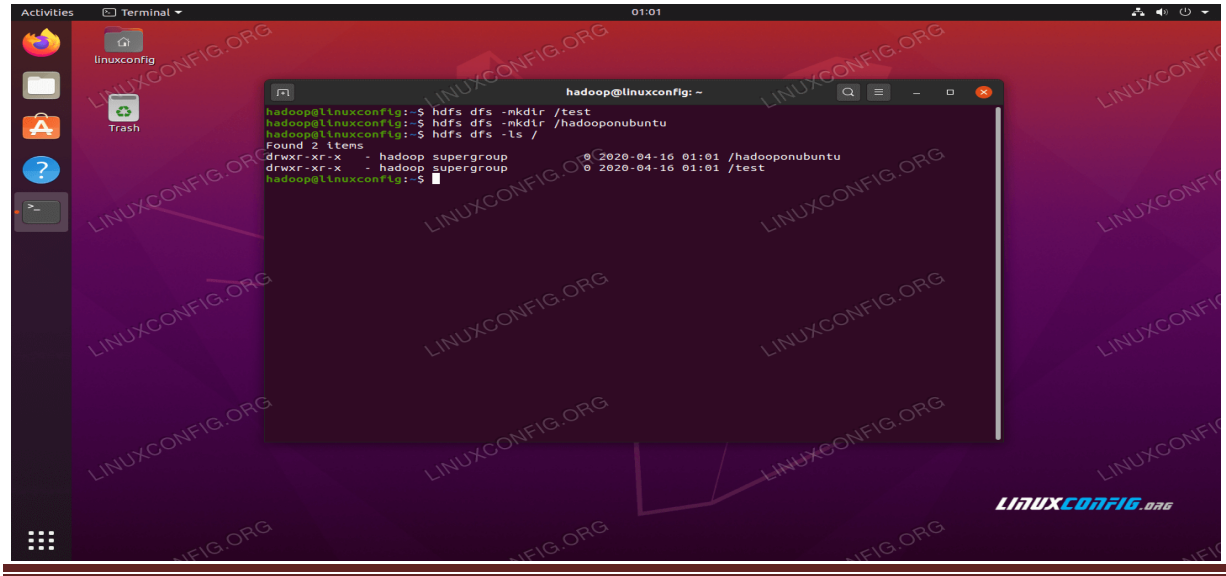
```
$ hdfs version
```



HDFS COMMAND LINE INTERFACE

The HDFS command line is used to access HDFS and to create directories or issue other commands to manipulate files and directories. Use the following command syntax to create some directories and list them:

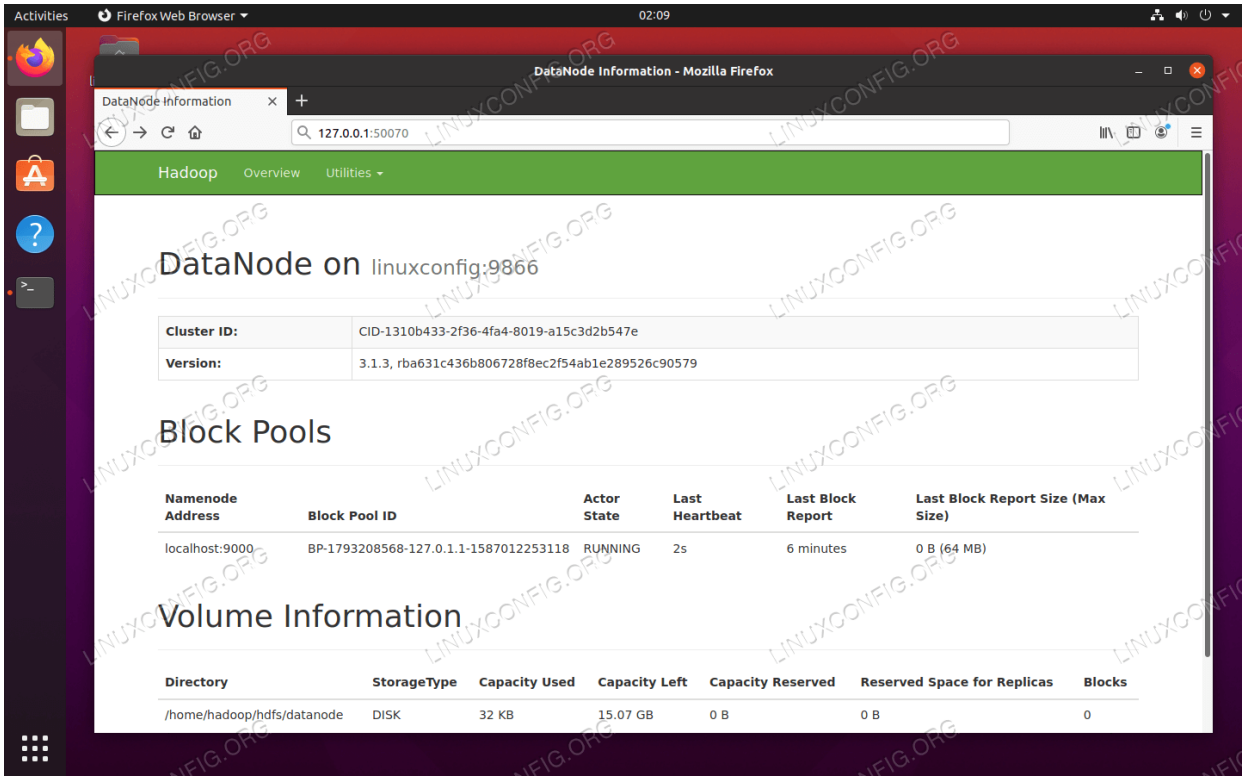
```
$ hdfs dfs -mkdir /test

$ hdfs dfs -mkdir /hadooponubuntu

$ hdfs dfs -ls /
```



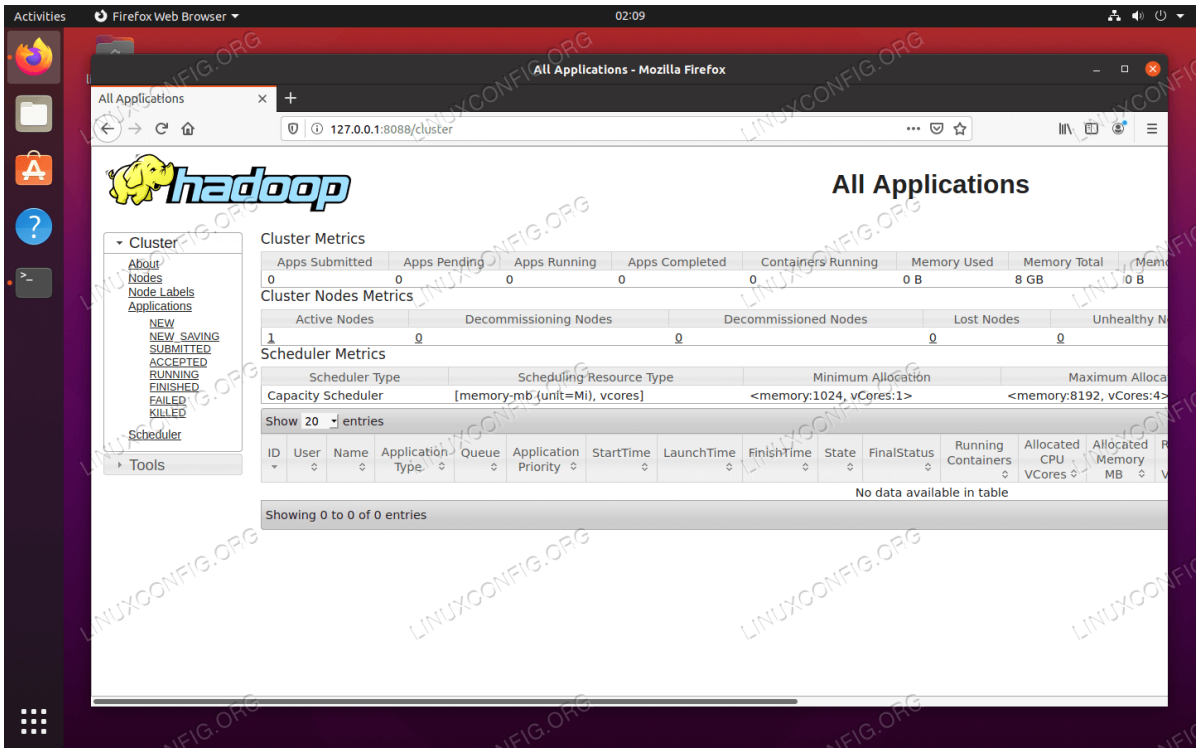
ACCESS THE NAMENODE AND YARN FROM BROWSER

You can access both the Web UI for NameNode and YARN Resource Manager via any browser of your choice, such as Mozilla Firefox or Google Chrome.

For the NameNode Web UI, navigate to **http://HADOOP-HOSTNAME-OR-IP:50070**



To access the YARN Resource Manager web interface, which will display all currently running jobs on the Hadoop cluster, navigate to **http://HADOOP-HOSTNAME-OR-IP:8088**



(II) KNOWING THE DIFFERENCING BETWEEN SINGLE NODE CLUSTERS AND MULTI-NODE CLUSTERS

Single node cluster : By default, Hadoop is configured to run in a non-distributed or standalone mode, as a single Java process. There are no daemons running and everything runs in a single JVM instance. HDFS is not used.

Pseudo-distributed or multi-node cluster: The Hadoop daemons run on a local machine, thus simulating a cluster on a small scale. Different Hadoop daemons run in different JVM instances, but on a single machine. HDFS is used instead of local FS

ACCESSING WEB-UI AND THE PORT NUMBER

That's how Spark reports that the web UI (which is known as SparkUI internally) is bound to the **port 4040**. As long as the Spark application is up and running, you can access the web UI at http://10.0.2.15:4040.

INSTALLING AND ACCESSING THE ENVIRONMENTS SUCH AS HIVE AND SQOOP

Step 1) Create a Sqoop directory by using the command **mkdir sqoop** so that we can download Apache Sqoop.

Step 2) Download the stable version of Apache Sqoop (ie Apache Sqoop 1.4.7 in the year 2022) Website URL https://archive.apache.org/dist/sqoop/1.4.7/

wget https://archive.apache.org/dist/sqoop/1.4.7/sqoop-1.4.7.bin__hadoop-2.6.0.tar.gz

Step 3) Unzip the downloaded file using the tar command

tar -xvzf sqoop-1.4.7.bin_hadoop-2.6.0.tar.gz

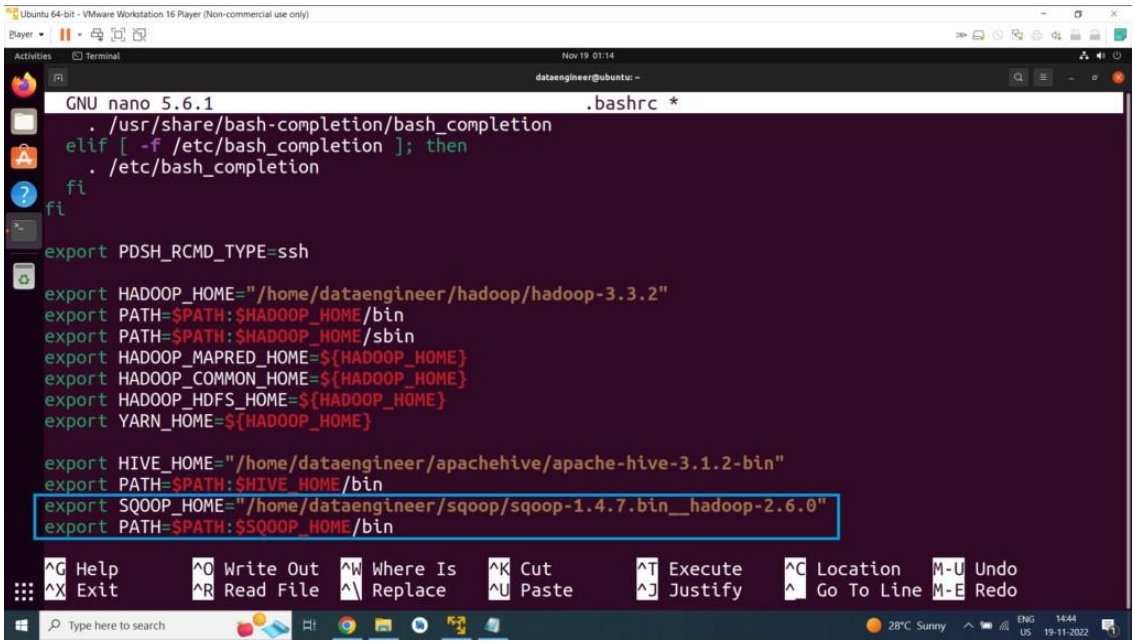
Step 4) Edit the .bashrc file by using the command

nano .bashrc

Step 5) Enter the following commands below in bashrc file and save it

export SQOOP_HOME=”/home/dataengineer/sqoop/sqoop-1.4.7.bin_hadoop-2.6.0″

export PATH=$PATH:$SQOOP_HOME/bin

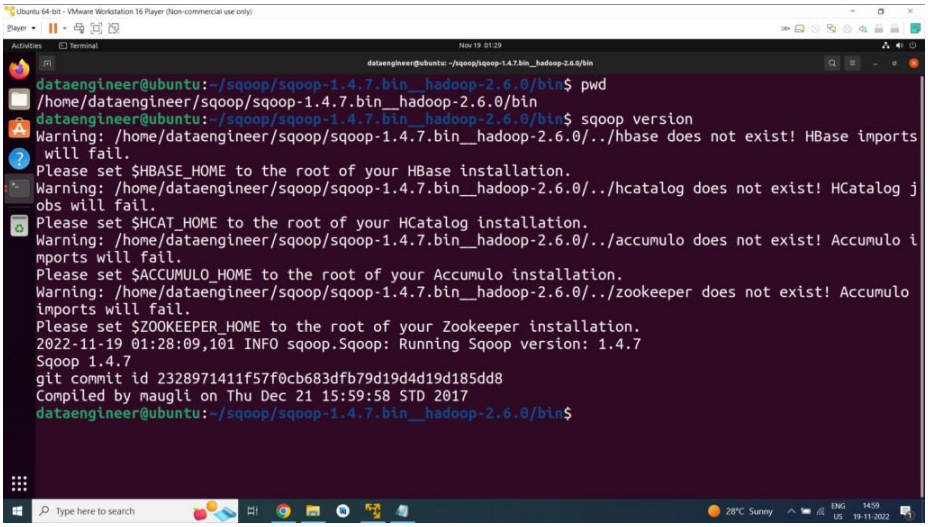


Step 6) Execute the below command on the command prompt so bashrc gets activated.

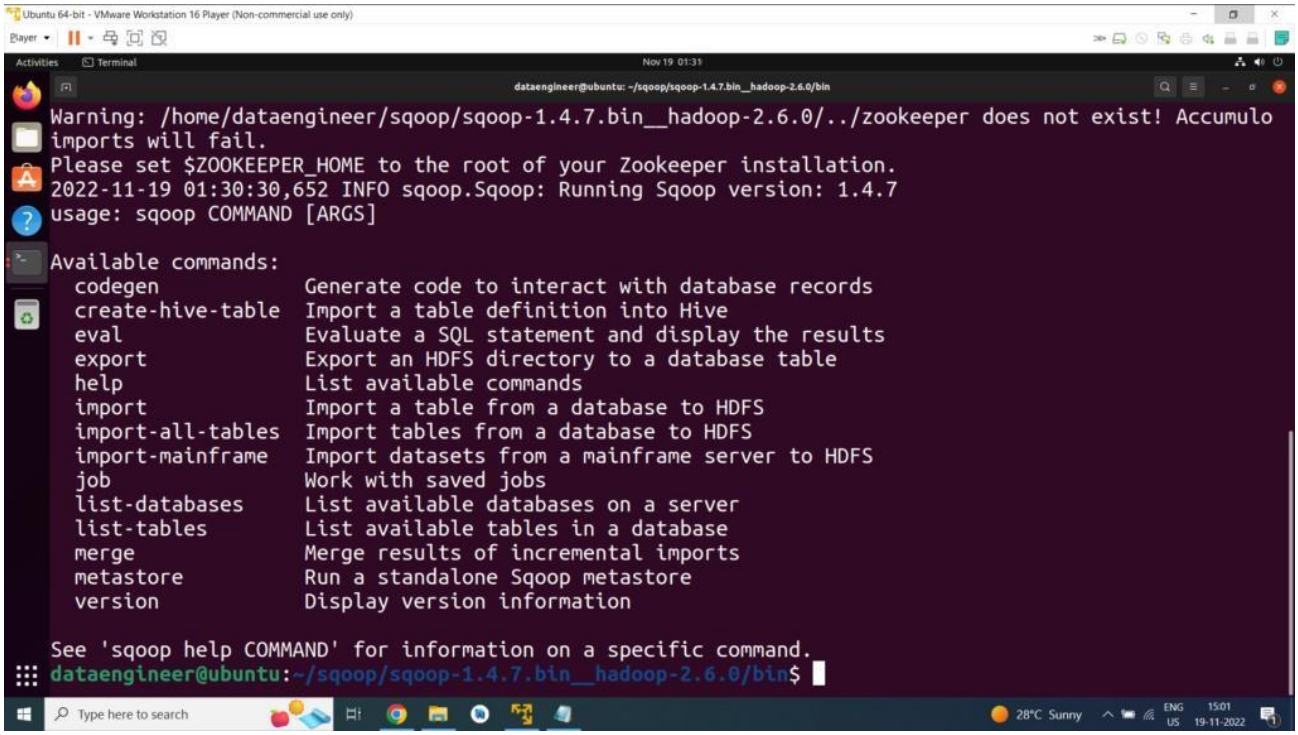
source ~/.bashrc

Step 7) Check the installed sqoop version using the below command

sqoop version



Step8) Type '**sqoop help**' to know the sqoop commands



MODULE-3

File management tasks & Basic linux commands

HDFS is the primary or major component of the Hadoop ecosystem which is responsible for storing large data sets of structured or unstructured data across various nodes and thereby maintaining the metadata in the form of log files. To use the HDFS commands, first you need to start the Hadoop services using the following command:

```
sbin/start-all.sh
```

To check the Hadoop services are up and running use the following command:

```
jps
```

```
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ jps
2546 SecondaryNameNode
2404 DataNode
2295 NameNode
2760 ResourceManager
2874 NodeManager
4251 Jps
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$
```

Commands:

1. **ls:** This command is used to list all the files. Use *lsr* for recursive approach. It is useful when we want a hierarchy of a folder. **Syntax:**

```
bin/hdfs dfs -ls  <path>
```

Example:

```
bin/hdfs dfs -ls /
```

It will print all the directories present in HDFS. bin directory contains executables so, *bin/hdfs* means we want the executables of hdfs particularly *dfs*(Distributed File System) commands.

```
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -ls /
19/01/31 10:35:11 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
Found 4 items
-rw-r--r--   1 suraj supergroup   13965969 2019-01-31 00:13 /input
drwxr-xr-x   - suraj supergroup          0 2019-01-31 01:30 /output
drwx------   - suraj supergroup          0 2019-01-31 00:15 /tmp
drwxr-xr-x   - suraj supergroup          0 2019-01-30 23:44 /user
```

2. **mkdir**: To create a directory. In Hadoop *dfs* there is no home directory by default. So let's first create it. **Syntax:**

```
bin/hdfs dfs -mkdir <folder name>
```

creating home directory:

```
hdfs/bin -mkdir /user

hdfs/bin -mkdir /user/username -> write the username of your computer
```

Example:

```
bin/hdfs dfs -mkdir  /geeks  =>  '/' means absolute path

bin/hdfs dfs -mkdir  geeks2  =>   Relative path -> the folder will be
                                  created relative to the home
directory.
```

```
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -mkdir /geeks
19/01/31 10:53:43 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -ls /
19/01/31 10:53:56 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
Found 5 items
drwxr-xr-x   - suraj supergroup          0 2019-01-31 10:53 /geeks
-rw-r--r--   1 suraj supergroup   13965969 2019-01-31 00:13 /input
drwxr-xr-x   - suraj supergroup          0 2019-01-31 01:30 /output
drwx------   - suraj supergroup          0 2019-01-31 00:15 /tmp
drwxr-xr-x   - suraj supergroup          0 2019-01-30 23:44 /user
```

```
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -mkdir geeks2
19/01/31 10:59:33 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -lsr /user
lsr: DEPRECATED: Please use 'ls -R' instead.
19/01/31 10:59:51 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
drwxr-xr-x   - suraj supergroup          0 2019-01-31 10:59 /user/suraj
-rw-r--r--   1 suraj supergroup      28093 2019-01-31 00:10 /user/suraj/AFINN-111.txt
drwxr-xr-x   - suraj supergroup          0 2019-01-31 10:59 /user/suraj/geeks2
drwxr-xr-x   - suraj supergroup          0 2019-01-30 23:47 /user/suraj/insideUserChecking
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$
```

3. **touchz**: It creates an empty file. **Syntax:**

```
bin/hdfs dfs  -touchz  <file_path>
```

Example:

```
bin/hdfs dfs -touchz  /geeks/myfile.txt
```

```
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -touchz /geeks/myfile.txt
19/01/31 11:10:31 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -lsr /geeks
lsr: DEPRECATED: Please use 'ls -R' instead.
19/01/31 11:10:48 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
-rw-r--r--   1 suraj supergroup          0 2019-01-31 11:10 /geeks/myfile.txt
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$
```

4. copyFromLocal (or) put:

To copy files/folders from local file system to hdfs store. This is the most important command. Local filesystem means the files present on the OS.

Syntax:

```
bin/hdfs dfs -copyFromLocal <local file path>  <dest(present on hdfs)>
```

Example: Let's suppose we have a file *AI.txt* on Desktop which we want to copy to folder *geeks* present on hdfs.

```
bin/hdfs dfs -copyFromLocal ../Desktop/AI.txt /geeks
```

(OR)

```
bin/hdfs dfs -put ../Desktop/AI.txt /geeks
```

```
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -put ../Desktop/AI.txt /geeks
19/01/31 11:31:02 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -lsr /geeks
lsr: DEPRECATED: Please use 'ls -R' instead.
19/01/31 11:31:21 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
-rw-r--r--   1 suraj supergroup        205 2019-01-31 11:31 /geeks/AI.txt
-rw-r--r--   1 suraj supergroup          0 2019-01-31 11:10 /geeks/myfile.txt
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$
```

5. cat:

To print file contents. **Syntax:**

```
bin/hdfs dfs -cat <path>
```

Example:

// print the content of AI.txt present

// inside geeks folder.

```
bin/hdfs dfs -cat /geeks/AI.txt ->
```

```
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -cat /geeks/AI.txt
19/01/31 11:33:25 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
In computer science, artificial intelligence, sometimes called machine intelligence, is intelligence demonstrated by machines, in contrast to t
he natural intelligence displayed by humans and other animals
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$
```

6. copyToLocal (or) get:

To copy files/folders from hdfs store to local file system.

Syntax:

```
bin/hdfs dfs -copyToLocal  <<srcfile(on hdfs)> <local file dest>
```

Example:

```
bin/hdfs dfs -copyToLocal  /geeks   ../Desktop/hero
```

(OR)

```
bin/hdfs dfs -get /geeks/myfile.txt  ../Desktop/hero
```

```
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -get /geeks/myfile.txt  ../Desktop/hero
19/01/31 11:43:34 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ ls ../Desktop/hero
myfile.txt
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$
```

Note: Observe that we don't write *bin/hdfs* while checking the things present on local filesystem.

7. moveFromLocal: This command will move file from local to hdfs.

Syntax:

```
bin/hdfs dfs -moveFromLocal <local src>   <dest(on hdfs)>
```

Example:

```
bin/hdfs dfs -moveFromLocal  ../Desktop/cutAndPaste.txt   /geeks
```

```
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -moveFromLocal ../Desktop/cutAndPaste.txt /geeks
19/01/31 12:38:38 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -ls /geeks
19/01/31 12:38:56 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
Found 3 items
-rw-r--r--   1 suraj supergroup        205 2019-01-31 11:31 /geeks/AI.txt
-rw-r--r--   1 suraj supergroup         96 2019-01-31 12:38 /geeks/cutAndPaste.txt
-rw-r--r--   1 suraj supergroup          0 2019-01-31 11:10 /geeks/myfile.txt
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ ls ../Desktop
AFINN-111.txt  Deploy_hadoop_on_a_single_node_cluster_v02 (1).pdf  hero                 sentiment.jar
AI.txt         FlumeData.1440939532959                             json-simple-1.1.1.jar  worldBank
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$
```

8. cp: This command is used to copy files within hdfs. Lets copy folder *geeks* to *geeks_copied*.

Syntax:

```
bin/hdfs dfs -cp  <src(on hdfs)>  <dest(on hdfs)>
```

Example:

```
bin/hdfs -cp /geeks  /geeks_copied
```

```
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -mkdir /geeks_copied
19/01/31 12:46:03 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -cp /geeks  /geeks_copied
19/01/31 12:46:26 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -ls /geeks_copied
19/01/31 12:47:10 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
Found 1 items
drwxr-xr-x   - suraj supergroup          0 2019-01-31 12:46 /geeks_copied/geeks
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$
```

9. mv: This command is used to move files within hdfs. Lets cut-paste a file *myfile.txt* from *geeks* folder to *geeks_copied*.

Syntax:

bin/hdfs dfs -mv <src(on hdfs)> <src(on hdfs)>

Example:

bin/hdfs -mv /geeks/myfile.txt /geeks_copied

```
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -ls /geeks
19/01/31 12:48:42 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
Found 3 items
-rw-r--r--   1 suraj supergroup        205 2019-01-31 11:31 /geeks/AI.txt
-rw-r--r--   1 suraj supergroup         96 2019-01-31 12:38 /geeks/cutAndPaste.txt
-rw-r--r--   1 suraj supergroup          0 2019-01-31 11:10 /geeks/myfile.txt
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -mv /geeks/myfile.txt  /geeks_copied
19/01/31 12:49:27 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -ls /geeks
19/01/31 12:49:42 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
Found 2 items
-rw-r--r--   1 suraj supergroup        205 2019-01-31 11:31 /geeks/AI.txt
-rw-r--r--   1 suraj supergroup         96 2019-01-31 12:38 /geeks/cutAndPaste.txt
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -ls /geeks_copied
19/01/31 12:50:00 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
Found 2 items
drwxr-xr-x   - suraj supergroup          0 2019-01-31 12:46 /geeks_copied/geeks
-rw-r--r--   1 suraj supergroup          0 2019-01-31 11:10 /geeks_copied/myfile.txt
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$
```

10. rmr: This command deletes a file from HDFS *recursively*. It is very useful command when you want to delete a *non-empty directory*.

Syntax:

bin/hdfs dfs -rmr <filename/directoryName>

Example:

bin/hdfs dfs -rmr /geeks_copied ->

It will delete all the content inside the directory then the directory itself.

```
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -ls /geeks_copied
19/01/31 12:58:16 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
Found 2 items
drwxr-xr-x   - suraj supergroup          0 2019-01-31 12:46 /geeks_copied/geeks
-rw-r--r--   1 suraj supergroup          0 2019-01-31 11:10 /geeks_copied/myfile.txt
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -rmr /geeks_copied
rmr: DEPRECATED: Please use 'rm -r' instead.
19/01/31 12:58:40 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
19/01/31 12:58:42 INFO fs.TrashPolicyDefault: Namenode trash configuration: Deletion interval = 0 minutes, Emptier interval = 0 minutes.
Deleted /geeks_copied
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -ls /geeks_copied
19/01/31 12:59:35 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
ls: `/geeks_copied': No such file or directory
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$
```

11. du: It will give the size of each file in directory.

Syntax:

```
bin/hdfs dfs -du  <dirName>
```

Example:

```
bin/hdfs dfs -du /geeks
```

```
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -du /geeks
19/01/31 13:01:57 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
205  205  /geeks/AI.txt
96   96   /geeks/cutAndPaste.txt
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$
```

12. dus:: This command will give the total size of directory/file.

Syntax:

```
bin/hdfs dfs -dus  <dirName>
```

Example:

```
bin/hdfs dfs -dus /geeks
```

```
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -dus /geeks
dus: DEPRECATED: Please use 'du -s' instead.
19/01/31 13:02:34 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
301  301  /geeks
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$
```

13. stat: It will give the last modified time of directory or path. In short it will give stats of the directory or file.

Syntax:

```
bin/hdfs  dfs -stat    <hdfs file>
```

Example:

```
bin/hdfs dfs -stat /geeks
```

```
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -stat /geeks
19/01/31 13:03:39 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
2019-01-31 07:19:29
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$
```

14. setrep: This command is used to change the replication factor of a file/directory in HDFS. By default it is 3 for anything which is stored in HDFS (as set in hdfs *core-site.xml*).

Example 1: To change the replication factor to 6 for *geeks.txt* stored in HDFS.

```
bin/hdfs dfs -setrep -R -w 6 geeks.txt
```

Example 2: To change the replication factor to 4 for a directory *geeksInput* stored in HDFS.

```
bin/hdfs dfs -setrep -R  4 /geeks
```

Note: The **-w** means wait till the replication is completed. And **-R** means recursively, we use it for directories as they may also contain many files and folders inside them.

```
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -setrep -R  4 /geeks
19/01/31 13:05:31 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
Replication 4 set: /geeks/AI.txt
Replication 4 set: /geeks/cutAndPaste.txt
suraj@suraj:~/hadoop-2.5.0-cdh5.3.2$ bin/hdfs dfs -ls /geeks
19/01/31 13:05:50 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applic
able
Found 2 items
-rw-r--r--   4 suraj supergroup        205 2019-01-31 11:31 /geeks/AI.txt
-rw-r--r--   4 suraj supergroup         96 2019-01-31 12:38 /geeks/cutAndPaste.txt
```

Note: There are more commands in HDFS but we discussed the commands which are commonly used when working with Hadoop. You can check out the list of *dfs* commands using the following command:

```
bin/hdfs dfs
```

```
Usage: hadoop fs [generic options]
        [-appendToFile <localsrc> ... <dst>]
        [-cat [-ignoreCrc] <src> ...]
        [-checksum <src> ...]
        [-chgrp [-R] GROUP PATH...]
        [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
        [-chown [-R] [OWNER][:[GROUP]] PATH...]
        [-copyFromLocal [-f] [-p] <localsrc> ... <dst>]
        [-copyToLocal [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-count [-q] <path> ...]
        [-cp [-f] [-p | -p[topax]] <src> ... <dst>]
        [-createSnapshot <snapshotDir> [<snapshotName>]]
        [-deleteSnapshot <snapshotDir> <snapshotName>]
        [-df [-h] [<path> ...]]
        [-du [-s] [-h] <path> ...]
        [-expunge]
        [-get [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-getfacl [-R] <path>]
        [-getfattr [-R] {-n name | -d} [-e en] <path>]
        [-getmerge [-nl] <src> <localdst>]
        [-help [cmd ...]]
        [-ls [-d] [-h] [-R] [<path> ...]]
        [-mkdir [-p] <path> ...]
        [-moveFromLocal <localsrc> ... <dst>]
        [-moveToLocal <src> <localdst>]
        [-mv <src> ... <dst>]
        [-put [-f] [-p] <localsrc> ... <dst>]
        [-renameSnapshot <snapshotDir> <oldName> <newName>]
        [-rm [-f] [-r|-R] [-skipTrash] <src> ...]
        [-rmdir [--ignore-fail-on-non-empty] <dir> ...]
        [-setfacl [-R] [{-b|-k} {-m|-x <acl_spec>} <path>]|[--set <acl_spec> <path>]]
        [-setfattr {-n name [-v value] | -x name} <path>]
        [-setrep [-R] [-w] <rep> <path> ...]
        [-stat [format] <path> ...]
        [-tail [-f] <file>]
        [-test -[defsz] <path>]
        [-text [-ignoreCrc] <src> ...]
```

MODULE-4

MAP-REDUCING

(i) Definition of Map-reduce

(ii) Its stages and terminologies

(iii) Word-count program to understand map-reduce (Mapper phase, Reducer phase, Driver code)

Definition of Map-reduce

It is a framework in which we can write applications to run huge amount of data in parallel and in large cluster of commodity hardware in a reliable manner.

stages and terminologies

Different Phases of MapReduce:-

MapReduce model has three major and one optional phase.

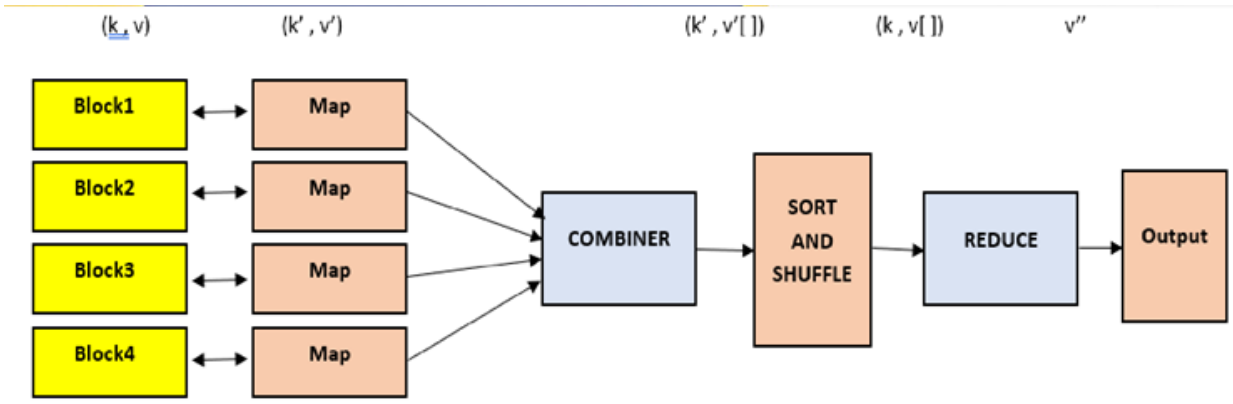
- Mapping
- Shuffling and Sorting
- Reducing
- Combining

Mapping :- It is the first phase of MapReduce programming. Mapping Phase accepts key-value pairs as input as (k, v), where the key represents the Key address of each record and the value represents the entire record content.The output of the Mapping phase will also be in the key-value format (k', v').

Shuffling and Sorting :- The output of various mapping parts (k', v'), then goes into Shuffling and Sorting phase. All the same values are deleted, and different values are grouped together based on same keys. The output of the Shuffling and Sorting phase will be key-value pairs again as key and array of values (k, v[ ]).

Reducer :- The output of the Shuffling and Sorting phase (k, v[]) will be the input of the Reducer phase. In this phase reducer function's logic is executed and all the values are Collected against their corresponding keys. Reducer stabilize outputs of various mappers and computes the final output.

Combining :- It is an optional phase in the MapReduce phases . The combiner phase is used to optimize the performance of MapReduce phases. This phase makes the Shuffling and Sorting phase work even quicker by enabling additional performance features in MapReduce phases.



Inputs and Outputs

The MapReduce framework operates exclusively on <key, value> pairs, that is, the framework views the input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job, conceivably of different types.

The key and value classes have to be serializable by the framework and hence need to implement the Writable interface. Additionally, the key classes have to implement the WritableComparable interface to facilitate sorting by the framework.

Input and Output types of a MapReduce job:

(input) <k1, v1> -> **map** -> <k2, v2> -> **combine** -> <k2, v2> -> **reduce** -> <k3, v3> (output)

Example: WordCount v1.0

Before we jump into the details, lets walk through an example MapReduce application to get a flavour for how they work.

WordCount is a simple application that counts the number of occurrences of each word in a given input set.

This works with a local-standalone, pseudo-distributed or fully-distributed Hadoop installation (Single Node Setup).

SOURCE CODE

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

 public static class TokenizerMapper
     extends Mapper<Object, Text, Text, IntWritable>{

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(Object key, Text value, Context context
                  ) throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
 }

 public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values,
                     Context context
                     ) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
```

```
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
 }

 public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = Job.getInstance(conf, "word count");
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

USAGE

Assuming environment variables are set as follows:

```
export JAVA_HOME=/usr/java/default
export PATH=${JAVA_HOME}/bin:${PATH}
export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
```

Compile WordCount.java and create a jar:

```
$ bin/hadoop com.sun.tools.javac.Main WordCount.java
$ jar cf wc.jar WordCount*.class
```

Assuming that:

- /user/joe/wordcount/input - input directory in HDFS
- /user/joe/wordcount/output - output directory in HDFS

Sample text-files as input:

```
$ bin/hadoop fs -ls /user/joe/wordcount/input/
/user/joe/wordcount/input/file01
/user/joe/wordcount/input/file02

$ bin/hadoop fs -cat /user/joe/wordcount/input/file01
Hello World Bye World

$ bin/hadoop fs -cat /user/joe/wordcount/input/file02
Hello Hadoop Goodbye Hadoop
```

Run the application:

```
$ bin/hadoop jar wc.jar WordCount /user/joe/wordcount/input /user/joe/wordcount/output
```

Output:

```
$ bin/hadoop fs -cat /user/joe/wordcount/output/part-r-00000
Bye 1
Goodbye 1
Hadoop 2
Hello 2
World 2
```

Running wordcount example with -libjars, -files and -archives:

bin/hadoop jar hadoop-mapreduce-examples-<ver>.jar wordcount -files cachefile.txt -libjars mylib.jar -archives myarchive.zip input output

Here, myarchive.zip will be placed and unzipped into a directory by the name “myarchive.zip”.

Users can specify a different symbolic name for files and archives passed through -files and -archives option, using #.

For example,

```
bin/hadoop jar hadoop-mapreduce-examples-<ver>.jar wordcount -files
dir1/dict.txt#dict1,dir2/dict.txt#dict2 -archives mytar.tgz#tgzdir input output
```

Here, the files dir1/dict.txt and dir2/dict.txt can be accessed by tasks using the symbolic names dict1 and dict2 respectively. The archive mytar.tgz will be placed and unarchived into a directory by the name "tgzdir".

Applications can specify environment variables for mapper, reducer, and application master tasks by specifying them on the command line using the options -Dmapreduce.map.env, -Dmapreduce.reduce.env, and -Dyarn.app.mapreduce.am.env, respectively.

For example the following sets environment variables FOO_VAR=bar and LIST_VAR=a,b,c for the mappers and reducers,

```
bin/hadoop jar hadoop-mapreduce-examples-<ver>.jar wordcount -
Dmapreduce.map.env.FOO_VAR=bar -Dmapreduce.map.env.LIST_VAR=a,b,c -
Dmapreduce.reduce.env.FOO_VAR=bar -Dmapreduce.reduce.env.LIST_VAR=a,b,c input
output
```

Walk-through

The WordCount application is quite straight-forward.

```
public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedException {
  StringTokenizer itr = new StringTokenizer(value.toString());
  while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
  }
}
```

The Mapper implementation, via the map method, processes one line at a time, as provided by the specified TextInputFormat. It then splits the line into tokens separated by whitespaces, via the StringTokenizer, and emits a key-value pair of < <word>, 1>.

For the given sample input the first map emits:

```
< Hello, 1>
< World, 1>
< Bye, 1>
< World, 1>
```

The second map emits:

```
< Hello, 1>
< Hadoop, 1>
< Goodbye, 1>
< Hadoop, 1>
```

We'll learn more about the number of maps spawned for a given job, and how to control them in a fine-grained manner, a bit later in the tutorial.

```
job.setCombinerClass(IntSumReducer.class);
```

WordCount also specifies a combiner. Hence, the output of each map is passed through the local combiner (which is same as the Reducer as per the job configuration) for local aggregation, after being sorted on the *key*s.

The output of the first map:

```
< Bye, 1>
< Hello, 1>
< World, 2>
```

The output of the second map:

```
< Goodbye, 1>
< Hadoop, 2>
< Hello, 1>
public void reduce(Text key, Iterable<IntWritable> values,
                   Context context
                   ) throws IOException, InterruptedException {
  int sum = 0;
  for (IntWritable val : values) {
    sum += val.get();
  }
  result.set(sum);
  context.write(key, result);
}
```

The Reducer implementation, via the reduce method just sums up the values, which are the occurrence counts for each key (i.e. words in this example).

Thus the output of the job is:

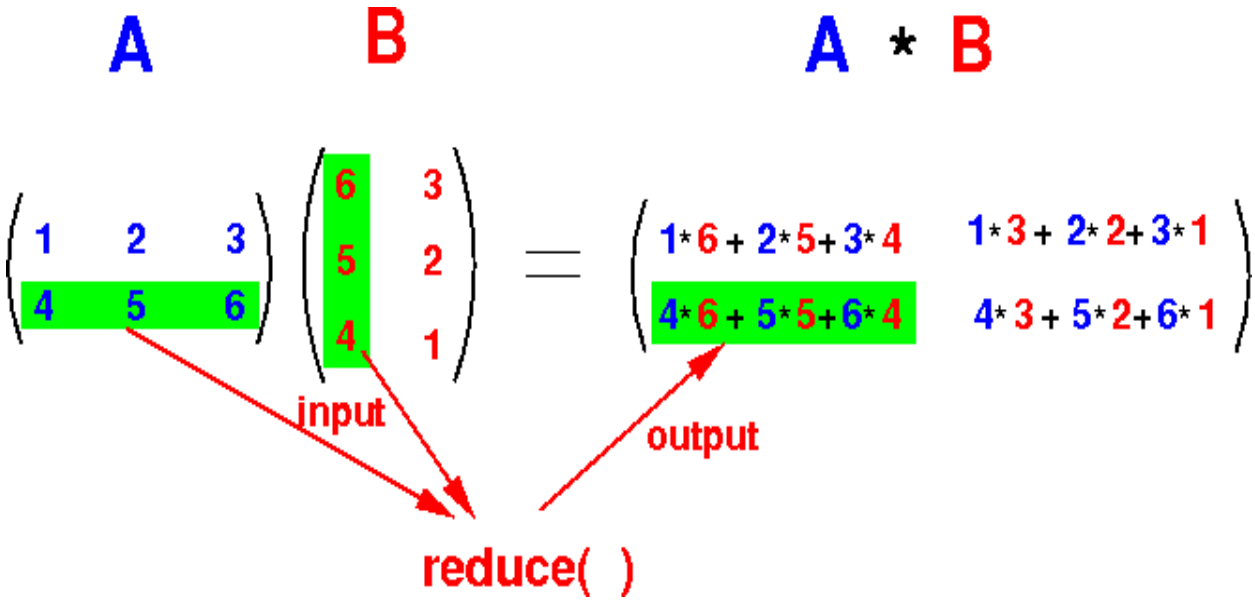
```
< Bye, 1>
< Goodbye, 1>
< Hadoop, 2>
< Hello, 2>
< World, 2>
```

The main method specifies various facets of the job, such as the input/output paths (passed via the command line), key/value types, input/output formats etc., in the Job. It then calls the job.waitForCompletion to submit the job and monitor its progress.

MODULE-5

IMPLEMENTING MATRIX-MULTIPLICATION WITH HADOOP MAP-REDUCE

In mathematics, matrix multiplication or the matrix product is a binary operation that produces a matrix from two matrices. The definition is motivated by linear equations and linear transformations on vectors, which have numerous applications in applied mathematics, physics, and engineering. In more detail, if A is an n × m matrix and B is an m × p matrix, their matrix product AB is an n × p matrix, in which the m entries across a row of A are multiplied with the m entries down a column of B and summed to produce an entry of AB. When two linear transformations are represented by matrices, then the matrix product represents the composition of the two transformations.



Map Reduce paradigm is the soul of distributed parallel processing in Big Data.

Before writing the code let's first create matrices and put them in HDFS.

- Create two files M1, M2 and put the matrix values. (sperate columns with spaces and rows with a line break)

```
For this example I am taking matrices as:
1 2 3          7  8
4 5 6          9  10
               11 12
```

- Put the above files to HDFS at location /user/clouders/matrices/

```
hdfs dfs –mkdir /user/cloudera/matrice
hdfs dfs –put /path/to/M1 /user/cloudera/matrices/
hdfs dfs –put /path/to/M2 /user/cloudera/matrices/
```

Let's start the code

We need to create two programs Mapper and Reducer.

Mapper.py

- First, define the dimensions of the matrices (m,n)

```
#!/usr/lib/python
import sys
m_r=2
m_c=3
n_r=3
n_c=2
i=0
```

Read each line i.e a row from stdin and split then to separate elements. Map int to each element as we read elements as string from stdin.

```
for line in sys.stdin:el=map(int,line.split())
```

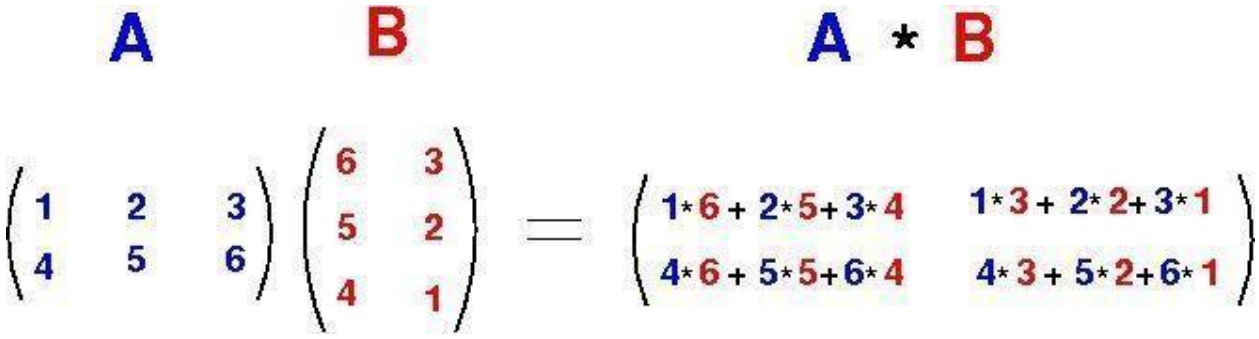
The mapper will first read the first matrix and then the second. To differentiate them we can keep a count i of the line number we are reading and the first m_r lines will belong to the first matrix.

```
if(i<m_r):
     for j in range(len(el)):
          for k in range(n_c): print "{0}\t{1}\t{2}\t{3}".format(i, k,
j, el[j])
 else:
     for j in range(len(el)):
          for k in range(m_r): print "{0}\t{1}\t{2}\t{3}".format(k, j,
i-m_r, el[j])
 i=i+1
```

Now comes the crucial part, printing the key value. We need to think of a key which will group elements that need to be multiplied, elements that need to be summed and elements that belong to the same row.

{0} {1} {2} are the part of key and {3} is the value.

To understand how I assigned a key, let's refer to the below image.



{0} {1} {2} actually represents the position of element from A or B to A*B

- {0} is the row position of the element
- {1} is the column position of the element
- {2} is the position of the element in addition. (like 1, 6 are at position 0 in addition and 2,5 are at position 1)

We can see that A's element is repeated B's number of column times i.e. 2 and B's element is repeated A's number of row times i.e. 2.

In the program

- i is used to iterate through each row
- j is used to iterate through each column

- k is used to iterate through each duplicate produced

For each element in matrix A:

- Element remains in same row, therefore {0}=i
- Element is duplicated and distributed to each column, therefore, column pos in A*B = Duplication order of element i.e. {1}=k
- As you can see in the picture, the position of the element, in addition, is the same as it's column's number therefore {2}=j

For each element in matrix B:

- Elements remain in the same column, therefore {1}=j
- Element is duplicated and distributed to each row, therefore, row pos in A*B = Duplication order of element i.e {0}=k
- As you can see in the picture, the position of the element, in addition, is the same as it's row's position therefore {2}=i-m_r

Output of Mapper.py

```
[cloudera@quickstart matrix-mu
0	0	0	1
0	1	0	1
0	0	1	2
0	1	1	2
0	0	2	3
0	1	2	3
1	0	0	4
1	1	0	4
1	0	1	5
1	1	1	5
1	0	2	6
1	1	2	6
0	0	0	7
1	0	0	7
0	1	0	8
1	1	0	8
0	0	1	9
1	0	1	9
0	1	1	10
1	1	1	10
0	0	2	11
1	0	2	11
0	1	2	12
1	1	2	12
```

If you will look closely you will realize that elements with the same key (first 3 numbers are key), will get multiplied. Elements with the same first two numbers of the key are part of the same sum and elements with same first num of key belong to the same row.

After mapper produces output, Hadoop will sort by key and provide it to reducer.py
\

Reducer.py

Our reducer program will get sorted mapper result which will look like this.

```
0	0	0	1
0	0	0	7
0	0	1	2
0	0	1	9
0	0	2	11
0	0	2	3
0	1	0	1
0	1	0	8
0	1	1	10
0	1	1	2
0	1	2	12
0	1	2	3
1	0	0	4
1	0	0	7
1	0	1	5
1	0	1	9
1	0	2	11
1	0	2	6
1	1	0	4
1	1	0	8
1	1	1	10
1	1	1	5
1	1	2	12
1	1	2	6
```

If you look closely at the output and image of matrix multiplication, you will realize:

- Every 2 numbers need to be multiplied
- Every m_c multiplied results need to get summed
- Every n_c summed result belong to the same row
- There will be m_r number of rows

- After the above observation, the reducer code seems easier.

```
#!/usr/lib/python
import sys
m_r=2
m_c=3
n_r=3
n_c=2matrix=[]
for row in range(m_r):
   r=[]
   for col in range(n_c):
      s=0
      for el in range(m_c):
         mul=1
         for num in range(2):
            line=sys.stdin.readline()
            n=map(int,line.split('\t'))[-1]
            mul*=n
         s+=mul
      r.append(s)
    matrix.append(r)
print('\n'.join([str(x) for x in matrix]))
```

Running the Map-Reduce Job on Hadoop

You can run the map reduce job and view the result by the following code (considering you have already put input files in HDFS)

```
$ chmod +x ~/Desktop/mr/matrix-mul/Mapper.py$ chmod +x ~/Desktop
/mr/matrix-mul/Reducer.py$ hadoop jar /usr/lib/hadoop-mapreduce
/hadoop-streaming.jar \
> -input /user/cloudera/matrices/ \
> -output /user/cloudera/mat_output \
> -mapper ~/Desktop/mr/matrix-mul/Mapper.py \
> -reducer ~/Desktop/mr/matrix-mul/Reducer.py
```

This will take some time as Hadoop do its mapping and reducing work. After the successful completion of the above process view the output by:

```
hdfs dfs -cat /user/cloudera/mat_output/*
```

Above command should output the resultant matrix

```
[cloudera@quickstart matrix-mul]$ hdfs dfs -cat /user/cloudera/mat_output/*
[14, 245]
[313, 77]
[cloudera@quickstart matrix-mul]$
```

This above code is not limited to any size. We can multiply matrices of any valid size by changing input and dimensions in the code.

MODULE-6

COMPUTE AVERAGE SALARY AND TOTAL SALARY BY GENDER FOR AN ENTERPRISE.

Problem statement:- Compute total and average salary of organization XYZ and group by based on sex(male or female).

Input data are in text file as tab separated. Schema of input data is - sex at position 4th and salary at 9th position. Download sample input file.

100 Steven King **M** SKING 515.123.4567 17-JUN-03 AD_PRES **25798.9** 90

Expected output:-

F Total: 291800.0 :: Average: 7117.073

M Total: 424363.34 :: Average: 6333.7812

We can think of this problem in terms of database SQL query as "SELECT SUM(SALARY), AVG(SALARY) FROM EMPLOYEES1 GROUP BY SEX" and same can be solved by HQL in hive.In the context of map/reduce, we have to write mapper(map method) and reducer (reduce method) class.

In **map method**, process input file line by line, split the given input line and extract sex and salary. Write extracted sex and salary in context object. Output of mapper is key as sex(M or F) and value as salary list of each employee as

<M sal1, sal2 ,sal3 ,.....>

<F sal1, sla2, sal3,. ..... >

In **reduce method**, salary list is iterated , total and average is computed. Total and average salary is written in context as Text with sex M or F.

Note:- In between map and reduce task, hadoop framework perform shuffle and sort based on key value. It can be verified by the output of this map/reduce program.In output file, record corresponding for M followed by for F (F come first in lexicographical order).

Sample Code:-

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
```

```
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;


/**
* @author http://www.devinline.com
*/
public class AverageAndTotalSalaryCompute {
/*
 * data schema(tab separated) :-100 Steven King M SKING 515.123.4567
 * 17-JUN-03 AD_PRES 25798.9 90 Sex at position 4th and salary at 9th
 * position
 */
public static class MapperClass extends
  Mapper<LongWritable, Text, Text, FloatWritable> {
 public void map(LongWritable key, Text empRecord, Context con)
   throws IOException, InterruptedException {
  String[] word = empRecord.toString().split("\\t");
  String sex = word[3];
  try {
   Float salary = Float.parseFloat(word[8]);
   con.write(new Text(sex), new FloatWritable(salary));
  } catch (Exception e) {
   e.printStackTrace();
  }
 }
}

public static class ReducerClass extends
  Reducer<Text, FloatWritable, Text, Text> {
 public void reduce(Text key, Iterable<FloatWritable> valueList,
   Context con) throws IOException, InterruptedException {
  try {
   Float total = (float) 0;
   int count = 0;
   for (FloatWritable var : valueList) {
    total += var.get();
    System.out.println("reducer " + var.get());
    count++;
   }
   Float avg = (Float) total / count;
   String out = "Total: " + total + " :: " + "Average: " + avg;
```

```
   con.write(key, new Text(out));
  } catch (Exception e) {
   e.printStackTrace();
  }
 }
}

public static void main(String[] args) {
 Configuration conf = new Configuration();
 try {
  Job job = Job.getInstance(conf, "FindAverageAndTotalSalary");
  job.setJarByClass(AverageAndTotalSalaryCompute.class);
  job.setMapperClass(MapperClass.class);
  job.setReducerClass(ReducerClass.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(FloatWritable.class);
  // Path p1 = new Path(args[0]);
  // Path p2 = new Path(args[1]);
  // FileInputFormat.addInputPath(job, p1);
  // FileOutputFormat.setOutputPath(job, p2);
  Path pathInput = new Path(
    "hdfs://192.168.213.133:54310/user/hduser1/employee_records.txt");
  Path pathOutputDir = new Path(
  "hdfs://192.168.213.133:54310/user/hduser1/testfs/output_mapred00");
  FileInputFormat.addInputPath(job, pathInput);
  FileOutputFormat.setOutputPath(job, pathOutputDir);
  System.exit(job.waitForCompletion(true) ? 0 : 1);
 } catch (IOException e) {
  e.printStackTrace();
 } catch (ClassNotFoundException e) {
  e.printStackTrace();
 } catch (InterruptedException e) {
  e.printStackTrace();
 }

}
}
```

In main method, Job object is using input and output directory of HDFS so start hadoop services (<hadoop_home>/sbin/start-all.sh). Copy input file from local file system to HDFS and change input location accordingly or uncomment 4 commented lines in main method and pass input and output information of local file system(comment HDFS file references).

Execute above program unit(Right click -> Run -> Run as hadoop) and verify the output using following commands.

```
hduser1@ubuntu:/usr/local/hadoop2.6.1/bin$ ./hadoop fs -cat
/user/hduser1/testfs/output_mapred00/part-r-00000
F Total: 291800.0 :: Average: 7117.073
M Total: 424363.34 :: Average: 6333.7812
```

Notice the output, *F record followed by M record* due to intermediate shuffle and sort operation by hadoop framework between map and reduce operation. Change input file mark some of row with sex value as T and execute above sample program unit and verify the output. It will appear like in lexicographically sorted order.

```
hduser1@ubuntu:/usr/local/hadoop2.6.1/bin$ ./hadoop fs -cat
/user/hduser1/testfs/output_mapred00/part-r-00000
F Total: 282200.0 :: Average: 7055.0
M Total: 412063.34 :: Average: 6438.4897
T Total: 21900.0 :: Average: 5475.0
```

MODULE-7

HIVE

(i) Creating hive tables (External and internal)

Table Operations such as **Creation, Altering, and Dropping** tables in Hive can be observed in this tutorial.

In the below screenshot, we are creating a table with columns and altering the table name.

1. Creating table guru_sample with two column names such as “empid” and “empname”

2. Displaying tables present in guru99 database

3. Guru_sample displaying under tables

4. Altering table “guru_sample” as “guru_sampleNew”

5. Again, when you execute “show” command, it will display the new name Guru_sampleNew

```
hive> create table guru_sample(empid int, empname string) ;   1
OK
Time taken: 1.635 seconds
hive> show tables;   2
OK
allstates
guru_sample   3
Time taken: 0.105 seconds, Fetched: 2 row(s)
hive> ALTER table guru_sample RENAME to guru_sampleNew;   4
OK
Time taken: 0.544 seconds
hive> show tables;
OK
allstates
guru_sampleNew   5
```

Dropping table guru_sampleNew:

```
hive> drop table guru_sampleNew;
OK
Time taken: 2.732 seconds
```

Dropping table

Table types and its Usage

Coming to **Tables** it's just like the way that we create in traditional Relational Databases. The functionalities such as filtering, joins can be performed on the tables.

Hive deals with two types of table structures like **Internal and External** tables depending on the loading and design of schema in Hive.

Internal tables

- Internal Table is tightly coupled in nature. In this type of table, first we have to create table and load the data.
- We can call this one as **data on schema**.
- By dropping this table, both data and schema will be removed.
- The stored location of this table will be at /user/hive/warehouse.

When to Choose Internal Table?

- If the processing data available in local file system
- If we want Hive to manage the complete lifecycle of data including the deletion

Sample code Snippet for Internal Table

1. To create the internal table

```
Hive>CREATE TABLE guruhive_internaltable (id INT,Name STRING);
        Row format delimited
        Fields terminated by '\t';
```

2. Load the data into internal table

```
Hive>LOAD DATA INPATH '/user/guru99hive/data.txt' INTO table guruhive_internaltable;
```

3. Display the content of the table

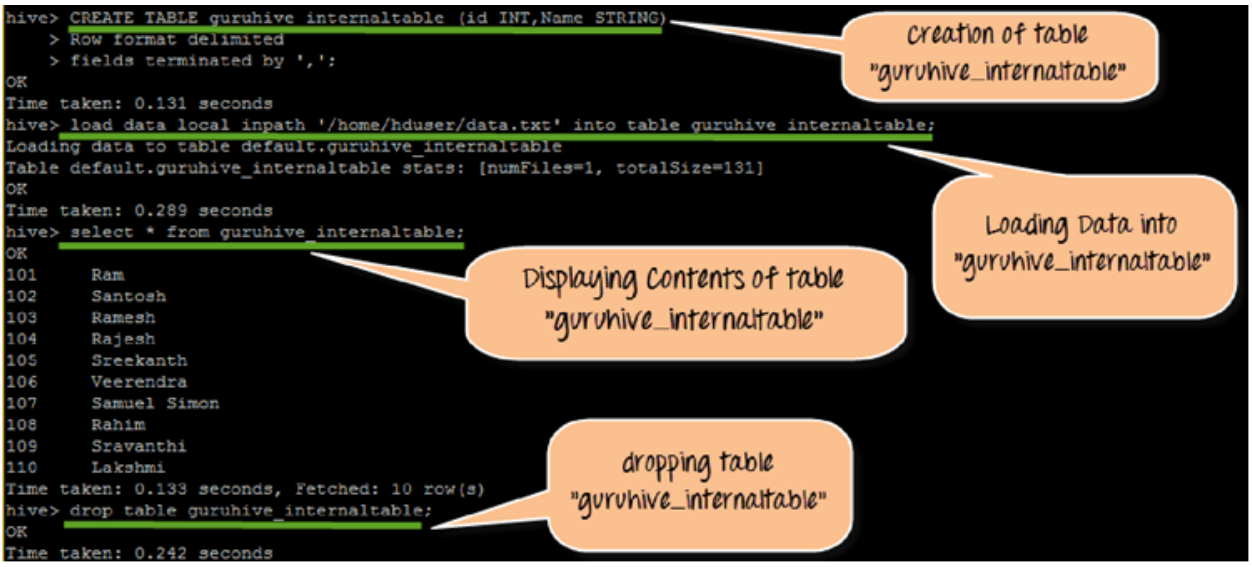
```
Hive>select * from guruhive_internaltable;
```

4. To drop the internal table

```
Hive>DROP TABLE guruhive_internaltable;
```

If you dropped the guruhive_internaltable, including its metadata and its data will be deleted from Hive.

From the following screenshot, we can observe the output



In above code and from screenshot we do following things,

- Create the internal table
- Load the data into internal table
- Display the content of the table
- To drop the internal table

External tables

- External Table is loosely coupled in nature. Data will be available in HDFS.The table is going to create on HDFS data.
- In other way, we can say like its creating **schema on data**.
- At the time of dropping the table it drops only schema, the data will be still available in HDFS as before.
- External tables provide an option to create multiple schemas for the data stored in HDFS instead of deleting the data every time whenever schema updates

When to Choose External Table?

- If processing data available in HDFS
- Useful when the files are being used outside of Hive

Sample code Snippet for External Table

1. Create External table

```
Hive>CREATE EXTERNAL TABLE guruhive_external(id INT,Name STRING)
        Row format delimited
        Fields terminated by '\t'
        LOCATION '/user/guru99hive/guruhive_external;
```

2. If we are not specifying the location at the time of table creation, we can load the data manually

```
Hive>LOAD DATA INPATH '/user/guru99hive/data.txt' INTO TABLE guruhive_external;
```

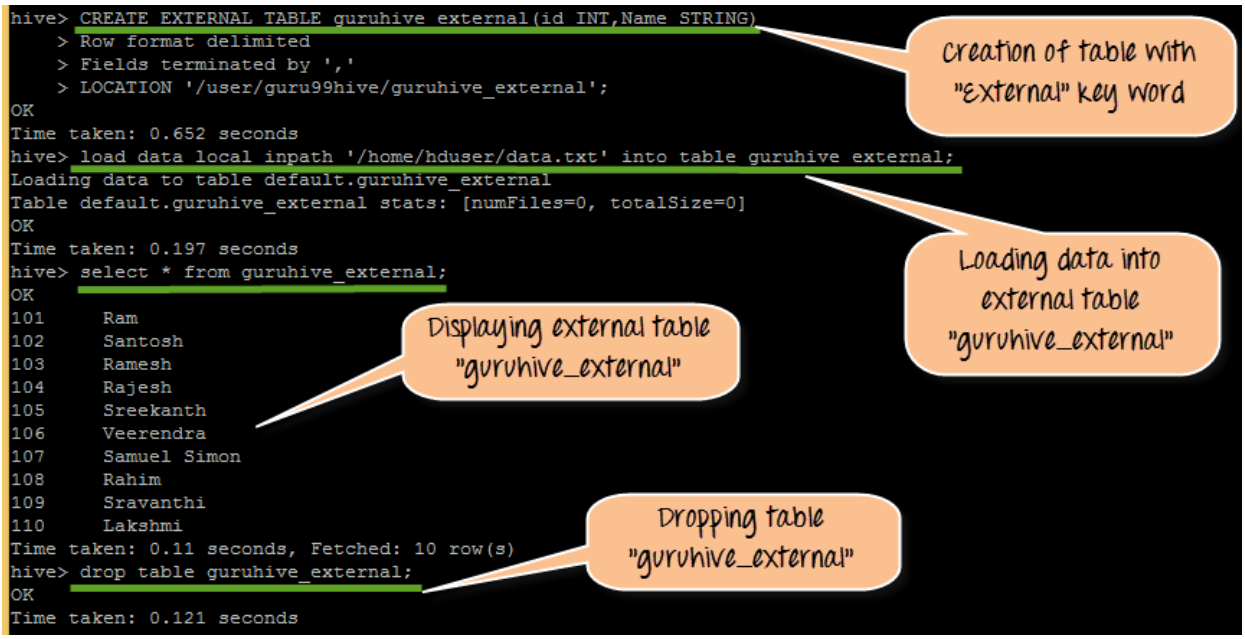
3. Display the content of the table

```
Hive>select * from guruhive_external;
```

4. To drop the internal table

```
Hive>DROP TABLE guruhive_external;
```

From the following screenshot, we can observe the output



In above code, we do following things

- Create the External table
- Load the data into External table
- Display the content of the table
- Dropping external tableDifference between Internal Vs External tables

Feature	Internal	External
Schema	Data on Schema	Schema on Data
Storage Location	/usr/hive/warehouse	HDFS location
Data availability	Within local file system	Within HDFS

ii) Loading data to external hive tables from sql tables

Loading data into Hive Table

We can load data into hive table in three ways.Two of them are DML operations of Hive.Third way is using hdfs command.If we have data in RDBMS system like Oracle,Mysql,DB2 or SQLServer we can import it using SQOOP tool.That Part we are not discussing now.

To Practice below commands ,create a table called Employee with below data

```
eno,ename,salary,dno

11,Balu,100000,15
12,Radha,120000,25
13,Nitya,150000,15
14,Sai Nirupam,120000,35
```

1. Using Insert Command

We can load data into a table using Insert command in two ways.One Using Values command and other is using queries.

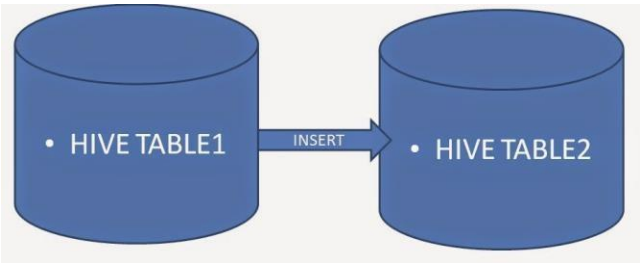
1.1 Using Values

Using Values command ,we can append more rows of data into existing table. for example ,to existing above employee table we can add extra row 15,Bala,150000,35 like below

```
Insert into table employee values (15,'Bala',150000,35)
```

After this You can run a select command to see newly added row.

1.2 Using Queries



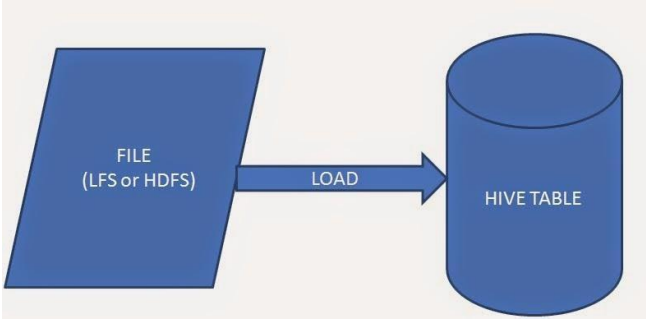
You can also upload query output into a table.for example Assume you have emp

table,from this, you can upload data into employee table like below

```
Insert into table employee Select * from emp where dno=45;
```

After this also You can fire select query to see uploaded rows.

2. Using Load



You can load data into a hive table using Load statement in two ways.
One is from local file system to hive table and other is from HDFS to Hive table.

2.1 From LFS to Hive Table

Assume we have data like below in LFS file called /data/empnew.csv.

```
15,Bala,150000,35
```

Now We can use load statement like below.

```
Load data local inpath '/data/empnew.csv' into table emp
```

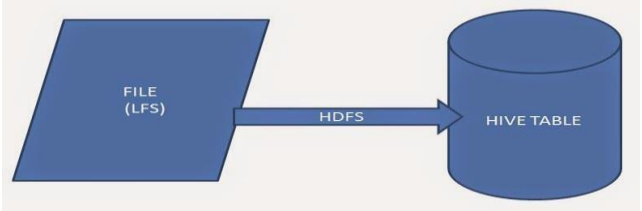
2.2 From HDFS to Hive Table

if we do not use local keyword ,it assumes it as a HDFS Path.

```
Load data  inpath '/data/empnew.csv' into table emp
```

After these two statements you can fire a select query to see loaded rows into table.

3. Using HDFS command



Assume You have data in a local file,You can simply upload data using hdfs commands.

run describe command to get the location of table like below.

```
describe formatted employee;
```

It will display Location of the table ,Assume You got location as /data/employee, you can upload data into table by using one of below commands.

```
hadoop fs -put /path/to/localfile /Data/employee
```

```
hadoop fs -copyFromLocal /path/to/localfile /Data/employee
```

```
hadoop fs -moveFromLocal /path/to/localfile /Data/esmployee
```

(iii) Performing operations like filterations and updations

In Hive, you can perform operations like filtering and updating data using HiveQL, which is similar to SQL. Here's how you can perform these operations:

Filtering Data

To filter data in Hive, you use the **WHERE** clause in your **SELECT** statements. The **WHERE** clause allows you to specify conditions that must be met for rows to be included in the result set.

```
-- Selecting data with a filter condition
SELECT *
FROM  table_name
WHERE condition;
```

For example:

```
-- Selecting data from a table where age is greater than 30
SELECT *
FROM employees
WHERE age > 30;
```

Updating Data

Hive does not support traditional SQL-style **UPDATE** statements for modifying data in existing tables. However, you can achieve similar results by overwriting the data in a table using the **INSERT OVERWRITE** statement in combination with a **SELECT** statement that filters and transforms the data as needed.

For example, to update the values in a table based on certain conditions, you can create a new table with the updated data and then overwrite the original table with the new data.

```
-- Create a new table with updated values
CREATE TABLE updated_table AS
SELECT
  CASE
    WHEN condition THEN new_value
    ELSE old_value
  END AS column_name,
  other_columns
FROM original_table;
-- Overwrite the original table with the updated data
INSERT OVERWRITE TABLE original_table
SELECT * FROM updated_table;
```

For instance:

```
-- Example: Increment salary by 10% for employees with more than 5 years of experience
CREATE TABLE updated_employees AS
SELECT
  employee_id,
  name,
  CASE
```

```
    WHEN experience_years > 5 THEN salary * 1.1
    ELSE salary
  END AS updated_salary
FROM employees;
-- Overwrite the original employees table with the updated data
INSERT OVERWRITE TABLE employees
SELECT * FROM updated_employees;
```

Remember, this operation creates a new table and replaces the original one, which can be resource-intensive and may not be suitable for large datasets. Make sure to test these operations on smaller datasets first and consider their impact on performance and data integrity. Additionally, always have backups of your data before performing such operations.

(iv) Performing Join (inner, outer etc)

In Hive, you can perform various types of joins, including inner, outer, left, right, and full outer joins, using the **JOIN** clause in your queries. Here's how you can perform different types of joins in Hive:

Inner Join

An inner join returns rows from both tables that satisfy the join condition.

```
SELECT *
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

Left Outer Join

A left outer join returns all rows from the left table and the matched rows from the right table. If there is no match, NULL values are returned for the columns from the right table.

```
SELECT *
FROM table1
LEFT OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

Right Outer Join

A right outer join returns all rows from the right table and the matched rows from the left table. If there is no match, NULL values are returned for the columns from the left table.

```
SELECT *

FROM table1

RIGHT OUTER JOIN table2

ON table1.column_name = table2.column_name;
```

Full Outer Join

A full outer join returns all rows when there is a match in either the left or right table. If there is no match, NULL values are returned for the columns from the other table.

```
SELECT *

FROM table1

FULL OUTER JOIN table2

ON table1.column_name = table2.column_name;
```

Example

Here's an example of how you might use a left outer join in Hive:

```sql
-- Assuming we have two tables: employees and departments

-- Selecting all employees with their corresponding department names

SELECT e.employee_id, e.name, d.department_name

FROM employees e

LEFT OUTER JOIN departments d

ON e.department_id = d.department_id;
```

In this example, the **employees** table is the left table, and the **departments** table is the right table. We're joining them on the **department_id** column. This query will return all employees, along with their department names if they belong to a department. If an employee doesn't belong to any department (i.e., there's no matching record in the **departments** table), the **department_name** column will contain NULL for that employee.

(v) Writing User defined function on hive tables

In Hive, you can write User-Defined Functions (UDFs) to extend the functionality of Hive by defining custom functions that can be used in HiveQL queries. UDFs allow you to encapsulate custom logic and perform complex operations that are not provided by built-in Hive functions. Here's a general guide on how to write and use UDFs in Hive:

Steps to Write a UDF:

1. **Choose a Programming Language**: Hive UDFs can be written in Java, Python, or any other language supported by Hive.
2. **Define the UDF Class or Function**: Write your custom logic inside a class or function. The method within this class or function will be executed for each input row.
3. **Implement the Appropriate Hive UDF Interface**: Depending on the type of UDF you want to create (e.g., scalar, aggregate, or table-generating), you need to implement the corresponding interface provided by Hive. Commonly used interfaces include:
    - `org.apache.hadoop.hive.ql.exec.UDF`: For scalar functions.
    - `org.apache.hadoop.hive.ql.exec.UDAF`: For aggregate functions.
    - `org.apache.hadoop.hive.ql.exec.UDTF`: For table-generating functions.
4. **Package and Compile**: Package your UDF code into a JAR file if you're using Java. Make sure to include any dependencies that your UDF relies on.
5. **Register the UDF**: Before using the UDF in Hive, you need to register it using the `ADD JAR` command or `CREATE FUNCTION` statement.

Example (Java UDF):

Here's a simple example of a Java UDF that converts a string to uppercase:

```
import org.apache.hadoop.hive.ql.exec.UDF;

import org.apache.hadoop.io.Text;

public class UpperCaseUDF extends UDF {

    public Text evaluate(Text input) {

        if (input == null) return null;

        return new Text(input.toString().toUpperCase());

    }

}
```

Usage in Hive:

Assuming you've compiled the Java class and packaged it into a JAR file named `udf.jar`, you can register and use the UDF in Hive:

```
-- Register the JAR file containing the UDF

ADD JAR /path/to/udf.jar;

-- Create or replace the UDF function

CREATE OR REPLACE FUNCTION upper_case AS 'com.example.udf.UpperCaseUDF';

-- Use the UDF in a Hive query

SELECT upper_case(column_name) FROM table_name;
```

Note:

- Ensure that the UDF JAR file is available on all nodes of your Hive cluster.
- When writing UDFs, consider performance implications, especially for UDFs used in large-scale data processing.
- Test your UDFs thoroughly to ensure they behave as expected and handle edge cases appropriately.

MODULE 8.

Create a sql table of employees Employee table with id,designation Salary table (salary ,dept id) Create external table in hive with similar schema of above tables,Move data to hive using scoop and load the contents into tables,filter a new table and

write a UDF to encrypt the table with AES-algorithm, Decrypt it with key to show contents

1. Create SQL Tables:

```
-- Employee table
CREATE TABLE Employee (
    id INT,
    designation VARCHAR(50)
);
-- Salary table
CREATE TABLE Salary (
    salary DECIMAL(10, 2),
    dept_id INT
);
```

2. Create External Hive Tables:

```
-- Create external Hive tables with similar schema
        CREATE EXTERNAL TABLE IF NOT EXISTS Employee_hive (
            id INT,
            designation STRING
        )
        ROW FORMAT DELIMITED
        FIELDS TERMINATED BY ','
        LOCATION '/path/to/employee_hive';
```

```
CREATE EXTERNAL TABLE IF NOT EXISTS Salary_hive (
   salary DECIMAL(10, 2),
   dept_id INT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION '/path/to/salary_hive';
```

3. Move Data to Hive using Sqoop:

Assuming you have installed and configured Sqoop, you can use it to import data from your SQL tables to Hive.

```
Sh
sqoop import \
--connect jdbc:mysql://your_mysql_host/your_database \
--username your_username \
--password your_password \
--table Employee \
--hive-import \
--hive-table Employee_hive \
--target-dir /path/to/employee_hive \
--fields-terminated-by ',';


sqoop import \
--connect jdbc:mysql://your_mysql_host/your_database \
--username your_username \
--password your_password \
--table Salary \
```

```
--hive-import \

--hive-table Salary_hive \

--target-dir  /path/to/salary_hive  \

--fields-terminated-by  ',';
```

4. Filter Data into a New Table:
You can create a new table by filtering data from existing tables in Hive using HiveQL.

```
SQL

CREATE TABLE Filtered_Salary AS

SELECT *

FROM Salary_hive

WHERE salary > 50000;
```

5. Write a UDF to Encrypt and Decrypt Data:
For encryption and decryption, you'll need to implement custom UDFs. Below is a simplified example using Java for encryption and decryption using AES algorithm.

```
java

import javax.crypto.Cipher;

import javax.crypto.spec.SecretKeySpec;

import org.apache.commons.codec.binary.Base64;


public class AESEncryptionUDF {

    private static final String ALGORITHM = "AES";

    private static final byte[] KEY = "YourSecretKey".getBytes(); // Change this with your secret key

    public String encrypt(String value) throws Exception {

        SecretKeySpec keySpec = new SecretKeySpec(KEY, ALGORITHM);

        Cipher cipher = Cipher.getInstance(ALGORITHM);
```

```
        cipher.init(Cipher.ENCRYPT_MODE, keySpec);
        byte[] encryptedValue = cipher.doFinal(value.getBytes());
        return Base64.encodeBase64String(encryptedValue);
    }
    public String decrypt(String encryptedValue) throws Exception {
        SecretKeySpec keySpec = new SecretKeySpec(KEY, ALGORITHM);
        Cipher cipher = Cipher.getInstance(ALGORITHM);
        cipher.init(Cipher.DECRYPT_MODE, keySpec);
        byte[] decryptedValue = cipher.doFinal(Base64.decodeBase64(encryptedValue));
        return new String(decryptedValue);
    }
}
```

6. Register and Use the UDF in Hive:
Assuming you have compiled the Java code into a JAR named **udf.jar**:

```
sql
ADD JAR /path/to/udf.jar;
CREATE FUNCTION encrypt AS 'com.example.udf.AESEncryptionUDF' USING JAR 'udf.jar';
CREATE FUNCTION decrypt AS 'com.example.udf.AESEncryptionUDF' USING JAR 'udf.jar';
-- Usage Example
SELECT encrypt(salary) FROM Salary_hive;
SELECT decrypt(encrypted_column) FROM Encrypted_Salary_hive;
```

This is a simplified example. Make sure to handle exceptions, security, and other aspects properly in a real-world scenario. Additionally, ensure that you manage your encryption keys securely.

MODULE -9 Pyspark

(i) Pyspark Definition(Apache Pyspark) and difference between Pyspark, Scala, pandas

PySpark Definition:

PySpark is the Python API for Apache Spark, a fast and general-purpose distributed computing system. Spark provides in-memory computing capabilities for big data processing, making it efficient and fast for large-scale data processing tasks. PySpark enables Python developers to interface with Spark using Python, making it easier for Python users to leverage Spark's capabilities for data processing, machine learning, graph processing, and more.

PySpark provides high-level APIs in Python that make it simple to build parallel applications. It also integrates well with Python libraries and tools, allowing developers to use familiar Python constructs and libraries within Spark applications.

Difference between PySpark, Scala, and pandas:

1. **PySpark**:
   - PySpark is the Python API for Apache Spark.
   - It allows Python developers to interact with Spark, enabling them to write Spark applications using Python.
   - PySpark leverages the distributed computing capabilities of Spark to process large-scale data efficiently.
   - It provides high-level APIs in Python for tasks such as data manipulation, SQL queries, machine learning, graph processing, and streaming analytics.
2. **Scala**:
   - Scala is a general-purpose programming language that runs on the Java Virtual Machine (JVM).
   - Scala is the primary language used for writing Apache Spark applications. Spark itself is written in Scala.
   - Scala offers strong support for functional programming and object-oriented programming paradigms.
   - Spark's Scala API provides the most comprehensive and native way to interact with Spark. It often provides better performance compared to other APIs due to its close integration with the Spark core.
3. **pandas**:
   - pandas is a Python library for data manipulation and analysis.
   - It provides high-performance, easy-to-use data structures and data analysis tools for Python.
   - pandas is particularly well-suited for data exploration, cleaning, and analysis in a single-machine environment.

- It offers DataFrame objects, similar to those in R and SQL, which make it easy to work with structured data.

Key Differences:

- **Programming Language**: PySpark uses Python, Scala uses Scala, and pandas is a Python library.
- **Performance**: Scala generally offers better performance than PySpark due to its close integration with Spark's core. pandas, being a single-machine library, may not scale as well as PySpark or Scala for large-scale distributed data processing tasks.
- **Ease of Use**: pandas is well-known for its ease of use and intuitive API. PySpark and Scala, being distributed computing frameworks, may have a steeper learning curve for beginners.
- **Ecosystem**: PySpark and Scala are part of the Apache Spark ecosystem, which provides a wide range of libraries and tools for various data processing tasks. pandas, being a Python library, integrates well with the Python data science ecosystem, including libraries like NumPy, scikit-learn, and Matplotlib.

(ii) Pyspark files and class methods

In PySpark, you typically work with the **SparkSession** object to interact with Spark. This object provides methods to create DataFrame, perform transformations, and execute actions on distributed datasets. Additionally, you can define custom classes and methods to encapsulate functionality and organize your PySpark code.

Here's a brief overview of common files and class methods you might encounter when working with PySpark:

PySpark Files:

1. **Driver Program**:
   - This is typically a Python script or application that initializes the SparkSession and submits jobs to the Spark cluster.
2. **Python Modules**:
   - You can organize your PySpark code into Python modules (**.py** files) to separate concerns and improve code maintainability.

Class Methods in PySpark:

1. **SparkSession**:
   - **SparkSession.builder**: Used to create a new **SparkSession** instance.
   - **getOrCreate()**: Retrieves an existing **SparkSession** or creates a new one if none exists.
   - **appName()**: Sets a name for the Spark application.
   - **master()**: Specifies the Spark master URL (e.g., **local[*]**, **yarn**, **spark://host:port**).
2. **DataFrame**:

- **createDataFrame()**: Creates a DataFrame from data sources such as RDDs, lists, or pandas DataFrames.
- **select()**: Selects columns from a DataFrame.
- **filter()**: Filters rows based on a condition.
- **groupBy()**: Groups the DataFrame using specified columns.
- **orderBy()**: Sorts the DataFrame by specified columns.
- **join()**: Joins two DataFrames based on a join expression.
- **agg()**: Performs aggregation operations like **sum()**, **avg()**, **max()**, **min()**, etc.
- **write()**: Writes the DataFrame to external storage (e.g., Parquet, CSV, JDBC).

3. **RDD (Resilient Distributed Dataset)**:
   - **map()**: Transforms each element of the RDD using a specified function.
   - **filter()**: Filters elements of the RDD based on a condition.
   - **reduce()**: Aggregates the elements of the RDD using a specified function.
   - **collect()**: Retrieves all elements of the RDD to the driver program (use with caution for large datasets).
   - **saveAsTextFile()**: Saves the RDD to a text file.
4. **SparkContext**:
   - **parallelize()**: Creates an RDD from a Python collection (e.g., list, tuple).

Custom Classes and Methods:

1. **User-Defined Functions (UDFs)**:
   - You can define custom functions using regular Python syntax and apply them to DataFrame columns using **withColumn()** and **selectExpr()** methods.
2. **Helper Classes and Methods**:
   - You can define custom classes and methods to encapsulate functionality and improve code organization.
   - For example, you might define a class to handle data preprocessing tasks, feature engineering, or model evaluation.
3. **Custom Transformations**:
   - You can create custom transformations by defining functions that operate on RDDs or DataFrames.
   - These functions can encapsulate complex logic or data processing steps that are not provided by built-in PySpark functions.

When working with PySpark, it's important to understand the structure of your application, organize your code effectively, and leverage the rich set of APIs provided by PySpark to manipulate and analyze large-scale datasets efficiently.

(iii) get(file name)

In PySpark, there is no built-in function named **get(file name)**. However, you might be referring to different functions or methods depending on the context of your question. Let me outline a few possibilities:

Possibilities:

1. **spark.read.load()**:
   - In PySpark, to read data from a file, you typically use the **spark.read.load()** method, where you specify the file format and the file path.
   - For example, to read a CSV file named "example.csv":

```
python

df = spark.read.load("example.csv", format="csv", header=True, inferSchema=True)
```

Python Built-in open() Function:

- If you want to read a file directly using Python, you can use the built-in **open()** function.
- For example:

```
python

with open("file.txt", "r") as file:
    data = file.read()
```

sc.textFile() for RDDs:

- If you're working with RDDs (Resilient Distributed Datasets) in PySpark, you might use **sc.textFile()** to read a text file into an RDD.
- For example:

```
python

rdd = sc.textFile("file.txt")
```

Other Custom Functions:

- If you are referring to a custom function named **get(file name)**, it would depend entirely on how it's defined within your PySpark codebase. It's not a standard function in PySpark.

(iv) get root directory()

In PySpark, you can't directly obtain the root directory of your Spark application using a built-in method or function like **get_root_directory()**. The root directory might refer to different things depending on the context:

1. **Working Directory of the Spark Application**:
   - You can obtain the working directory of the Spark application using Python's os module.
   - This directory is where your Spark application is executed.
   - Example:

```
python

import os

root_directory = os.getcwd()

print("Root Directory:", root_directory)
```

Spark Configuration:

- You can access Spark configuration parameters to get relevant directories.
- For example, you can use **sparkContext.getConf()** to access Spark configuration parameters.
- Example:

```
Python

from pyspark import SparkContext

sc = SparkContext.getOrCreate()

spark_conf = sc.getConf()

spark_root_directory = spark_conf.get("spark.home")

print("Spark Root Directory:", spark_root_directory)
```

Path Handling with Hadoop Configuration:

- If you need to access Hadoop-related directories, you can use Hadoop configuration parameters.
- Example:

```
Python

from py4j.java_gateway import java_import

java_import(sc._gateway.jvm, "")

hadoop_conf =
sc._gateway.jvm.org.apache.hadoop.fs.FileSystem.get(sc._jsc.hadoopConfiguration())

hadoop_root_directory = hadoop_conf.getWorkingDirectory()

print("Hadoop Root Directory:", hadoop_root_directory)
```

Remember, the "root directory" might mean different things in different contexts, so make sure you choose the appropriate method based on your requirements.

MODULE -10
Pyspark -RDD'S

In PySpark, RDDs (Resilient Distributed Datasets) are the primary abstraction for working with distributed data. RDDs represent an immutable, distributed collection of objects that can be operated on in parallel across a cluster. They serve as the foundation for performing data processing tasks in Spark.

Here are some key characteristics of RDDs in PySpark:

1. **Resilient**: RDDs are resilient to failures because they can automatically recover from faults by recomputing lost partitions using lineage information.
2. **Distributed**: RDDs are distributed across the nodes in a Spark cluster. They are partitioned across multiple nodes, enabling parallel processing of data.
3. **Immutable**: RDDs are immutable, meaning once created, they cannot be changed. However, you can apply transformations to create new RDDs.
4. **Lazy Evaluation**: Transformations applied to RDDs are lazily evaluated, meaning Spark defers computation until an action is called. This allows Spark to optimize execution plans and minimize data shuffling.
5. **Fault-Tolerant**: RDDs are fault-tolerant because Spark tracks the lineage of transformations applied to RDDs. If a partition of an RDD is lost due to node failure, Spark can recompute it using the lineage information.
6. **Partitioned**: RDDs are divided into partitions, which are the basic units of parallelism in Spark. Each partition of an RDD can be processed independently on different nodes in the cluster.
7. **Functional API**: RDDs support functional-style operations such as **map**, **filter**, **reduce**, **flatMap**, **groupByKey**, **sortByKey**, and more. These operations enable data processing tasks like transformation, filtering, aggregation, and sorting.

Here's a simple example of creating an RDD and applying transformations in PySpark

```
from pyspark import SparkContext #

Create a SparkContext

sc = SparkContext("local", "RDD Example") #

Create an RDD from a list

data = [1, 2, 3, 4, 5]

rdd = sc.parallelize(data) #
```

```
squared_rdd = rdd.map(lambda x: x ** 2)

# Perform actions

print(squared_rdd.collect())  # Collect the results

# Stop the SparkContext

sc.stop()
```

(iii) parallelized collections (iv) external dataset (v) existing RDD's

In PySpark, you can create RDDs (Resilient Distributed Datasets) using various methods to parallelize existing Python collections, load data from external sources, or by transforming existing RDDs. Here are the primary ways to create RDDs in PySpark:

Parallelize a Python Collection:
You can create an RDD from an existing Python collection (e.g., list, tuple) using the **parallelize()** method of the SparkContext object.

```
python

from pyspark import SparkContext

# Create a SparkContext

sc = SparkContext("local", "RDD Creation")

# Create an RDD from a Python collection

data = [1, 2, 3, 4, 5]

rdd = sc.parallelize(data)

# Stop the SparkContext

sc.stop()
```

Read from External Data Sources:
You can create RDDs by loading data from external data sources such as text files, CSV files, JSON files, HDFS, HBase, Cassandra, etc. PySpark provides methods like **textFile()**, **wholeTextFiles()**, **sequenceFile()**, and **hadoopFile()** to read data from these sources.

```
python

from pyspark import SparkContext

# Create a SparkContext

sc = SparkContext("local", "RDD Creation")

# Read data from a text file and create an RDD

rdd = sc.textFile("file:///path/to/file.txt")

# Stop the SparkContext

sc.stop()
```

Transform Existing RDDs:
You can create new RDDs by applying transformations on existing RDDs. Transformations such as **map()**, **flatMap()**, **filter()**, **groupByKey()**, **reduceByKey()**, etc., can be applied to create new RDDs.

```
python

from pyspark import SparkContext

# Create a SparkContext

sc = SparkContext("local", "RDD Creation")

# Create an RDD from a Python collection

data = [1, 2, 3, 4, 5]

rdd1 = sc.parallelize(data)

# Apply a transformation to create a new RDD

rdd2 = rdd1.map(lambda x: x ** 2)

# Stop the SparkContext

sc.stop()
```

Spark RDD's operations (Count, foreach(), Collect, join,Cache()

Count:

- Returns the number of elements in the RDD.

python

```
count = rdd.count()
```

foreach():

- Applies a function to each element of the RDD without returning a new RDD.
- This is typically used for actions such as printing elements or updating external state.

```
python

def print_element(x):

    print(x)

rdd.foreach(print_element)
```

Collect:

- Retrieves all elements of the RDD as a local Python list.
- Use with caution for large datasets as it brings all data to the driver node.

```
python

collected_data = rdd.collect()
```

Join:

- Performs an inner join between two RDDs based on a common key.

```
python

rdd1 = sc.parallelize([(1, 'a'), (2, 'b'), (3, 'c')])

rdd2 = sc.parallelize([(1, 'x'), (2, 'y'), (4, 'z')])

joined_rdd = rdd1.join(rdd2)
```

Cache():

- Persist the RDD in memory for faster access in subsequent actions.
- This is particularly useful for iterative algorithms or when the RDD will be used multiple times.

```
python

rdd.cache()
```

Example illustrating these operations:

```python
from pyspark import SparkContext
# Create a SparkContext
sc = SparkContext("local", "RDD Operations")
# Create an RDD
rdd = sc.parallelize([1, 2, 3, 4, 5])
# Count
count = rdd.count()
print("Count:", count)
# foreach
def print_element(x):
    print(x)
rdd.foreach(print_element)
# Collect
collected_data = rdd.collect()
print("Collected Data:", collected_data)
# Join
rdd1 = sc.parallelize([(1, 'a'), (2, 'b'), (3, 'c')])
rdd2 = sc.parallelize([(1, 'x'), (2, 'y'), (4, 'z')])
joined_rdd = rdd1.join(rdd2)
print("Joined RDD:")
print(joined_rdd.collect())
# Cache
rdd.cache()
# Stop the SparkContext
sc.stop()
```

MODULE -11

Perform pyspark transformations

In PySpark, **map** and **flatMap** are transformation operations applied to RDDs (Resilient Distributed Datasets) that enable you to transform each element of an RDD into another element or sequence of elements. Both operations are functional in nature and apply a given function to each element of the RDD independently and in parallel across the distributed cluster.

Here's an explanation of **map** and **flatMap** in PySpark:

1. map Transformation:

The **map** transformation applies a specified function to each element of the RDD and returns a new RDD containing the transformed elements. The function passed to **map** operates on each element of the RDD independently.

Syntax:

```
python

new_rdd = rdd.map(lambda x: <transformation_function(x)>)
```

Example:

```
python

# Create an RDD

rdd = sc.parallelize([1, 2, 3, 4, 5])

# Apply map transformation to square each element

squared_rdd = rdd.map(lambda x: x ** 2)
```

In the example above, the **map** transformation squares each element of the RDD, resulting in a new RDD with squared elements.

2. flatMap Transformation:

The **flatMap** transformation is similar to **map**, but it can return multiple elements for each input element. It "flattens" the resulting sequence of elements into a single RDD. This is particularly useful when the transformation function generates zero, one, or multiple output elements for each input element.

Syntax:

```
python
```

```
new_rdd = rdd.flatMap(lambda x: <transformation_function(x)>)
```

Example:

```
python

# Create an RDD with lists of words

rdd = sc.parallelize(["hello world", "how are you"])

# Apply flatMap transformation to split words

flat_mapped_rdd = rdd.flatMap(lambda x: x.split())
```

In the example above, the **flatMap** transformation splits each string by whitespace, resulting in an RDD with individual words.

Differences:

- **map** preserves the structure of the RDD, whereas **flatMap** can "flatten" nested structures.
- **map** applies a one-to-one transformation, while **flatMap** can apply a one-to-many transformation.
- **flatMap** is often used for tokenization, splitting strings, or other scenarios where one input element generates multiple output elements.

Both **map** and **flatMap** are fundamental operations in PySpark for transforming data and building complex data processing pipelines. Understanding when to use each operation is crucial for effective data processing.

ii) to remove the words, which are not necessary to analyze this text.

to remove words that are not necessary for analysis from a text using PySpark, you can follow these general steps:

1. Tokenize the text: Split the text into individual words.
2. Filter out unnecessary words: Remove words that are not necessary for analysis, such as stopwords or specific words.
3. Join the remaining words back into a single text or store them in a data structure for further analysis.

Here's how you can implement these steps in PySpark:

```
python

from pyspark import SparkContext

# Create a SparkContext

sc = SparkContext("local", "Remove Unnecessary Words")

# Sample text

text = "This is a sample text containing some words that are not necessary for analysis."

# Create an RDD from the text

text_rdd = sc.parallelize([text])

# Tokenize the text and lowercase all words

words_rdd = text_rdd.flatMap(lambda line: line.lower().split())

# Define a list of stopwords or words to remove

stopwords = ["is", "a", "the", "that", "are", "not", "for", "some"]

# Filter out stopwords

filtered_words_rdd = words_rdd.filter(lambda word: word not in stopwords)

# Join the remaining words into a single string

filtered_text = " ".join(filtered_words_rdd.collect())

print("Filtered Text:", filtered_text)

# Stop the SparkContext

sc.stop()
```

In this example, we create an RDD from a sample text and then tokenize the text by splitting it into individual words. We then filter out the stopwords defined in the **stopwords** list using the **filter** transformation. Finally, we join the remaining words back into a single string.

iii) groupBy

n PySpark, the **groupBy** transformation is used to group the elements of an RDD based on a key. It groups elements that have the same key into a collection (an Iterable) of key-value pairs, where the key is the grouping key, and the value is an Iterable containing all the elements that share the same key.

Here's how you can use **groupBy** in PySpark:

```
Python

from pyspark import SparkContext

# Create a SparkContext

sc = SparkContext("local", "groupBy Example")

# Create an RDD with key-value pairs

data = [(1, 'a'), (2, 'b'), (1, 'c'), (2, 'd'), (3, 'e')]

rdd = sc.parallelize(data)

# Group elements by key

grouped_rdd = rdd.groupBy(lambda x: x[0])

# Iterate through each group

for key, values in grouped_rdd.collect():

    print("Key:", key)

    print("Values:", list(values))  # Convert Iterable to list for easier printing

# Stop the SparkContext

sc.stop()
```

In this example, we first create an RDD **rdd** containing key-value pairs. Then, we use the **groupBy** transformation to group the elements by the key (the first element of each tuple). The result is an RDD **grouped_rdd**, where each element is a key-value pair where the key is the grouping key, and the value is an Iterable containing all the elements that share the same key.

Finally, we iterate through each group in **grouped_rdd** using the **collect** action and print the key along with the list of values for each group.

(iv) What if we want to calculate how many times each word is coming in corpus ?

To calculate how many times each word appears in a corpus using PySpark, you can follow these general steps:

1. Tokenize the text: Split the text into individual words.

2. Map each word to a key-value pair, where the word is the key and the value is 1 (indicating one occurrence of the word).
3. Use the **reduceByKey** transformation to aggregate the counts for each word.
4. Optionally, filter out any unnecessary words, such as stopwords.

Here's how you can implement these steps in PySpark:

python

```
from pyspark import SparkContext

# Create a SparkContext

sc = SparkContext("local", "Word Count Example")

# Sample text corpus

corpus = [

    "This is a sample text containing some words.",

    "The words may appear multiple times in the text."

]

# Create an RDD from the corpus

corpus_rdd = sc.parallelize(corpus)

# Tokenize the text and lowercase all words

words_rdd = corpus_rdd.flatMap(lambda line: line.lower().split())

# Map each word to a key-value pair (word, 1)

word_count_rdd = words_rdd.map(lambda word: (word, 1))

# Reduce by key to aggregate the counts for each word

word_counts = word_count_rdd.reduceByKey(lambda a, b: a + b)

# Print the word counts

print("Word Counts:")

for word, count in word_counts.collect():

    print(f"{word}: {count}")
```

```
# Stop the SparkContext

sc.stop()
```

In this example, we create an RDD from a sample text corpus. We then tokenize the text by splitting it into individual words and convert each word into a key-value pair **(word, 1)** using the **map** transformation. Next, we use the **reduceByKey** transformation to aggregate the counts for each word. Finally, we collect and print the word counts.

How do I perform a task (say count the words 'spark' and 'apache' in rdd3) separatly on each partition and get the output of the task performed in these partition

To perform a task separately on each partition of an RDD and get the output of the task performed in each partition, you can use the **mapPartitions** transformation in PySpark. The **mapPartitions** transformation allows you to apply a function to each partition of the RDD independently.

Here's how you can use **mapPartitions** to count the occurrences of the words 'spark' and 'apache' separately on each partition:

In this example:

- We create an RDD **rdd** from the sample data.
- We define a function **count_words_in_partition** that takes an iterator over the elements of a partition, counts the occurrences of 'spark' and 'apache' in each partition, and yields the counts.
- We apply the **mapPartitions** transformation to the RDD to apply the **count_words_in_partition** function to each partition.
- Finally, we collect and print the counts per partition.

Using **mapPartitions**, you can perform customized operations on each partition of an RDD efficiently.

python

```
from pyspark import SparkContext

# Create a SparkContext

sc = SparkContext("local", "Word Count per Partition Example")

# Sample data

data = ['apache spark is a great tool', 'apache is a part of hadoop ecosystem', 'spark provides distributed computing']

# Create an RDD from the sample data

rdd = sc.parallelize(data, numSlices=2)  # numSlices specifies the number of partitions

# Function to count occurrences of 'spark' and 'apache' in each partition

def count_words_in_partition(iterator):

    count_spark = 0

    count_apache = 0

    for text in iterator:

        count_spark += text.count('spark')

        count_apache += text.count('apache')

    yield (count_spark, count_apache)

# Apply mapPartitions to count occurrences of 'spark' and 'apache' in each partition

counts_per_partition = rdd.mapPartitions(count_words_in_partition).collect()

# Print the counts per partition

for idx, counts in enumerate(counts_per_partition):

    print(f"Partition {idx + 1}: Spark Count = {counts[0]}, Apache Count = {counts[1]}")

# Stop the SparkContext

sc.stop()
```

(vi) unions of RDD (vii) join two pairs of RDD Based upon their key

In PySpark, you can combine the elements of two RDDs using the `union` transformation. The `union` transformation concatenates the elements of two RDDs into a new RDD without removing any duplicates.

Here's how you can use the `union` transformation to combine two RDDs:

```
python

from pyspark import SparkContext

# Create a SparkContext

sc = SparkContext("local", "RDD Union Example")

# Create RDD 1

rdd1 = sc.parallelize([1, 2, 3])

# Create RDD 2

rdd2 = sc.parallelize([3, 4, 5])

# Combine RDDs using union

union_rdd = rdd1.union(rdd2)

# Collect and print the elements of the union RDD

print("Union of RDDs:")

print(union_rdd.collect())

# Stop the SparkContext

sc.stop()
```

In this example:

- We create two RDDs, **rdd1** and **rdd2**, containing integer elements.
- We use the **union** transformation to combine the elements of **rdd1** and **rdd2** into a new RDD called **union_rdd**.
- The **collect** action is used to retrieve all elements of the **union_rdd** RDD and print them.

The output of the example will be:

```
csharp

Union of RDDs:

[1, 2, 3, 3, 4, 5]
```

join two pairs of RDD Based upon their key

To join two pairs of RDDs based on their keys in PySpark, you can use the **join** transformation. The **join** operation allows you to combine pairs of RDDs where the keys match. Here's how you can do it:

```
python

from pyspark import SparkContext

# Create a SparkContext

sc = SparkContext("local", "RDD Join Example")

# Create RDD 1 with key-value pairs

rdd1 = sc.parallelize([(1, 'a'), (2, 'b'), (3, 'c')])

# Create RDD 2 with key-value pairs

rdd2 = sc.parallelize([(1, 'x'), (2, 'y'), (4, 'z')])

# Perform inner join on RDDs based on keys

joined_rdd = rdd1.join(rdd2)

# Print the result of the join operation

print("Joined RDD:")

print(joined_rdd.collect())

# Stop the SparkContext

sc.stop()
```

In this example:

- We create two RDDs, **rdd1** and **rdd2**, each containing key-value pairs.
- We use the **join** transformation to perform an inner join on **rdd1** and **rdd2** based on their keys.
- The **join** transformation results in an RDD containing pairs of elements where the keys match.

- We collect and print the elements of the joined RDD.

The output of the example will be:

```
less

Joined RDD:

[(1, ('a', 'x')), (2, ('b', 'y'))]
```

As you can see, the elements from both RDDs are joined based on their keys. In this case, only the pairs with keys 1 and 2 are included in the joined RDD because they are present in both **rdd1** and **rdd2**.

MODULE-12.

Pyspark sparkconf-Attributes and applications

In PySpark, **SparkConf** is a configuration class that allows you to set various Spark properties and configurations for your Spark applications. It provides a way to customize the behavior of your Spark application and control aspects such as memory allocation, parallelism, logging, and more.

Here are some commonly used attributes of **SparkConf** and their applications:

1. **setAppName(appName)**:
   - Sets a name for your Spark application, which is displayed in the Spark web UI and logs.
   - Example: **conf.setAppName("MySparkApp")**
2. **setMaster(master)**:
   - Sets the master URL for your Spark application. It specifies the cluster manager to use (e.g., local, standalone, YARN, Mesos).
   - Example: **conf.setMaster("local[*]")**
3. **set("spark.executor.memory", memory)**:
   - Sets the amount of memory to allocate per executor.
   - Example: **conf.set("spark.executor.memory", "2g")**
4. **set("spark.executor.cores", cores)**:
   - Sets the number of cores to allocate per executor.
   - Example: **conf.set("spark.executor.cores", "4")**
5. **set("spark.executor.instances", instances)**:
   - Sets the number of executor instances to launch in the cluster.
   - Example: **conf.set("spark.executor.instances", "4")**
6. **set("spark.driver.memory", memory)**:
   - Sets the amount of memory to allocate for the driver program.
   - Example: **conf.set("spark.driver.memory", "1g")**
7. **set("spark.default.parallelism", parallelism)**:
   - Sets the default number of partitions to use for RDDs in operations like **parallelize** and **defaultParallelism**.
   - Example: **conf.set("spark.default.parallelism", "10")**
8. **set("spark.serializer", serializer)**:
   - Sets the serializer for RDD serialization.
   - Example: **conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")**
9. **set("spark.streaming.stopGracefullyOnShutdown", "true")**:
   - Enables graceful shutdown of streaming applications.
   - Example: **conf.set("spark.streaming.stopGracefullyOnShutdown", "true")**
10. **set("spark.sql.shuffle.partitions", partitions)**:

- Sets the number of partitions to use when shuffling data for joins or aggregations in Spark SQL.
- Example: **conf.set("spark.sql.shuffle.partitions", "200")**

11. **set("spark.ui.port", port)**:
    - Sets the port for the Spark web UI.
    - Example: **conf.set("spark.ui.port", "4040")**

These are just a few examples of the many configuration options available in PySpark's **SparkConf** class. By customizing these configurations, you can optimize the performance, resource utilization, and behavior of your Spark applications based on your specific requirements and the characteristics of your cluster.

(i) What is Pyspark sparkconf ()

The **SparkConf** class is used to create a configuration object that holds key-value pairs of Spark properties. These properties define the behavior of the Spark application when it runs on a cluster.

Here are some common uses of **SparkConf**:

1. **Setting Application Name**: You can set a name for your Spark application using **setAppName()**. This name will be displayed in the Spark web UI and logs.
2. **Setting Master URL**: You can specify the URL of the master node (e.g., local, standalone, YARN, Mesos) using **setMaster()**.
3. **Setting Executor and Driver Memory**: You can allocate memory to executors and the driver program using **set("spark.executor.memory", memory)** and **set("spark.driver.memory", memory)** respectively.
4. **Setting Executor Cores**: You can specify the number of cores per executor using **set("spark.executor.cores", cores)**.
5. **Setting Default Parallelism**: You can set the default number of partitions for RDDs using **set("spark.default.parallelism", parallelism)**.
6. **Setting Serializer**: You can specify the serializer for RDD serialization using **set("spark.serializer", serializer)**.
7. **Setting UI Port**: You can specify the port for the Spark web UI using **set("spark.ui.port", port)**.

These are just a few examples of the many configuration options available in PySpark's **SparkConf** class. By customizing these configurations, you can optimize the performance, resource utilization, and behavior of your Spark applications based on your specific requirements and the characteristics of your cluster.

Here's a basic example of how you can create a **SparkConf** object and set configurations:

```
python

from pyspark import SparkConf

# Create a SparkConf object

conf = SparkConf()

# Set configurations

conf.setAppName("MySparkApp")

conf.setMaster("local[*]")

conf.set("spark.executor.memory", "2g")

# Now, you can pass this SparkConf object when creating a SparkContext

from pyspark import SparkContext

sc = SparkContext(conf=conf)
```

In this example, we create a **SparkConf** object, set various configurations, and then pass this object to create a **SparkContext** (**sc**). The **SparkContext** will use the configurations defined in the **SparkConf** object.

ii) Using spark conf create a spark session to write a dataframe to read details in a c.s.v and later move that c.s.v to another location

the outlines the process of configuring a Spark application using **SparkConf** and creating a **SparkContext** with those configurations.

Let's break down the process step by step:

Create a SparkConf object:

```
python

from pyspark import SparkConf

conf = SparkConf()
```

Here, you import **SparkConf** from the **pyspark** module and create an instance of **SparkConf** named **conf**. At this point, **conf** is an empty configuration object.

Set various configurations:

```
conf.setAppName("MySparkApp")

conf.setMaster("local[*]")

conf.set("spark.executor.memory", "2g")
```

With **conf**, you set various configurations for your Spark application:

- **setAppName**: Sets the name of your Spark application to "MySparkApp".
- **setMaster**: Sets the master URL to "local[*]", which means Spark will run in local mode using all available CPU cores.
- **set("spark.executor.memory", "2g")**: Sets the amount of memory each executor can use to 2 gigabytes.

Create a SparkContext with the configured SparkConf:

```
python

from pyspark import SparkContext

sc = SparkContext(conf=conf)
```

Here, you import **SparkContext** from **pyspark** and create a **SparkContext** named **sc**. By passing the **conf** object as an argument to **SparkContext**, you ensure that the **SparkContext** will use the configurations defined in **conf**.

By following this pattern, you can customize and fine-tune various aspects of your Spark application's behavior, such as resource allocation, parallelism, and application name. This flexibility allows you to optimize your Spark application for different use cases and deployment environments.