GR20 Regulations

III B.Tech II Semester

BIG DATA ANALYTICS LAB

(GR20A3133)

Department of AIML Engineering

(Artificial Intelligence and Machine Learning)

GOKARAJU RANGARAJU

INSTITUTE OF ENGINEERING AND TECHNOLOGY

(Autonomous)

SYLLABUS

GOKARAJU RANGARAJU INSTITUTE OF ENGINEERING AND TECHNOLOGY

BIG DATA ANALYTICS LAB

Course Code: GR20A3133 **L/T/P/C: 0/0/3/1.5**
III Year II Semester

Course Objectives:

1. Provide the knowledge to setup a Hadoop Cluster.
2. Impart knowledge to develop programs using MapReduce.
3. Discuss Pig, PigLatin and HiveQL to process big data.
4. Present latest big data frameworks and applications using Spark
5. Integrate Hadoop with R (RHadoop) to process and visualize.

Course Outcomes:

1. Understand Hadoop working environment.
2. Apply Map Reduce programs for real world problems.
3. Implement scripts using Pig to solve real world problems.
4. Analyze queries using Hive to analyze the datasets
5. Understand spark working environment and integration with R

TASK 1
a) Understanding and using basic HDFS commands
b) Run a basic word count Map Reduce program to understand Map Reduce Paradigm

TASK 2
Write a Map Reduce program that mines weather data

TASK 3
Implement matrix multiplication with Hadoop Map Reduce.

TASK 4
Working with files in Hadoop file system: Reading, Writing andCopying

TASK 5
Write Pig Latin scripts sort, group, join, project, and filter your data.

TASK 6
Run the Pig Latin Scripts to find Word Count and max temp for each and every year.

TASK 7
Writing User Defined Functions/Eval functions for filtering unwanted data in Pig

TASK 8
Working with Hive QL, Use Hive to create, alter, and drop databases, tables, views, functions and indexes

TASK 9
Writing User Defined Functions in Hive

TASK 10
Understanding the processing of large dataset on Spark framework.

TASK 11
Ingesting structured and unstructured data using sqoop and flume.

TASK 12
Integrating Hadoop with other data analytic framework like R

Text Books

1. Tom White, “Hadoop: The Definitive Guide”, 4th Edition, O’Reilly Inc,2015.
2. Tanmay Deshpande, “Hadoop Real-World Solutions Cookbook”, 2ndEdition, Packt Publishing, 2016.

Reference

1. Edward Capriolo, Dean Wampler, and Jason Rutherglen, “Programming Hive”, O’Reilly Inc, 2012.
2. Vignesh Prajapati, “Big data Analytics with R and Hadoop”, Packt Publishing, 2013.

INDEX

TASK 1

a) Understanding and using basic HDFS commands

Apache Hadoop hadoop fs or hdfs dfs are file system commands to interact with HDFS, these commands are very similar to Unix Commands. Note that some Syntax and output formats may differ between Unix and HDFS Commands.
Hadoop is a open-source distributed framework that is used to store and process a large set of datasets. To store data, Hadoop uses HDFS, and to process data, it uses MapReduce & Yarn. In this article, I will mainly focus on Hadoop HDFS commands to interact with the files.

Hadoop provides two types of commands to interact with File System; hadoop fs or hdfs dfs. Major difference being hadoop commands are supported with multiple file systems like S3, Azure and many more.

Start Hadoop Services
In order to run hdfs dfs or hadoop fs commands, first, you need to start the Hadoop services by running the start-dfs.sh script from the Hadoop installation.

ubuntu@namenode:~$ start-dfs.sh
Starting namenodes on [namenode.socal.rr.com]
Starting datanodes
Starting secondary namenodes [namenode]
ubuntu@namenode:~$

Basic HDFS DFS Commands

Below are basic hdfs dfs or hadoop fs Commands.

COMMAND	DESCRIPTION
`-ls`	List files with permissions and other details
`-mkdir`	Creates a directory named path in HDFS
`-rm`	To Remove File or a Directory
`-rmr`	Removes the file that identified by path / Folder and subfolders
`-rmdir`	Delete a directory
`-put`	Upload a file / Folder from the local disk to HDFS
`-cat`	Display the contents for a file
`-du`	Shows the size of the file on hdfs.
`-dus`	Directory/file of total size
`-get`	Store file / Folder from HDFS to local file

`-getmerge`	Merge Multiple Files in an HDFS
`-count`	Count number of directory, number of files and file size
`-setrep`	Changes the replication factor of a file
`-mv`	HDFS Command to move files from source to destination
`-moveFromLocal`	Move file / Folder from local disk to HDFS
`-moveToLocal`	Move a File to HDFS from Local
`-cp`	Copy files from source to destination
`-tail`	Displays last kilobyte of the file
`-touch`	create, change and modify timestamps of a file
`-touchz`	Create a new file on HDFS with size 0 bytes
`-appendToFile`	Appends the content to the file which is present on HDF
`-copyFromLocal`	Copy file from local file system
`-copyToLocal`	Copy files from HDFS to local file system
`-usage`	Return the Help for Individual Command
`-checksum`	Returns the checksum information of a file
`-chgrp`	Change group association of files/change the group of a file or a path
`-chmod`	Change the permissions of a file
`-chown`	change the owner and group of a file
`-df`	Displays free space
`-head`	Displays first kilobyte of the file
`-Create Snapshots`	Create a snapshot of a snapshottable directory
`-Delete Snapshots`	Delete a snapshot of from a snapshottable directory
`-Rename Snapshots`	Rename a snapshot
`-expunge`	create new checkpoint
`-Stat`	Print statistics about the file/directory
`-truncate`	Truncate all files that match the specified file pattern to the specified length
`-find`	Find File Size in HDFS

ls – List Files and Folder

HDFS ls command is used to display the list of Files and Directories in HDFS, This ls command shows the files with permissions, user, group, and other details. For more information follow ls- List Files and Folder

```
$hadoop fs -ls
```

```
or
$hdfs dfs -ls
```

mkdir – Make Directory

HDFS mkdir command is used to create a directory in HDFS. By default, this directory would be owned by the user who is creating it. By specifying "/" at the beginning it creates a folder at root directory.

```
$hadoop fs -mkdir /directory-name
or
$hdfs dfs -mkdir /directory-name
```

rm – Remove File or Directory

HDFS rm command deletes a file and a directory from HDFS recursively.

```
$hadoop fs -rm /file-name
or
$hdfs dfs -rm /file-name
```

rmr – Remove Directory Recursively

Rmr command is used to deletes a file from Directory recursively, it is a very useful command when you want to delete a non-empty directory.

```
$hadoop fs -rmr /directory-name
or
$hdfs dfs -rmr /directory-name
```

rmdir – Delete a Directory

Rmdir command is used to removing directories only if they are empty.

```
$hadoop fs -rmdir /directory-name
or
$hdfs dfs -rmdir /directory-name
```

put – Upload a File to HDFS from Local

Copy file/folder from local disk to HDFS. On put command specifies the local-file-path where you wanted to copy from and then hdfs-file-path where you wanted to copy to on hdfs.

```
$ hadoop fs -put /local-file-path /hdfs-file-path
or
$ hdfs dfs -put /local-file-path /hdfs-file-path
```

cat – Displays the Content of the File

The cat command reads the specified file from HDFS and displays the content of the file on console or stdout.

```
$ hadoop fs -cat /hdfs-file-path
or
$ hdfs dfs -cat /hdfs-file-path
```

du – File Occupied in Disk

Du command is used to How much file Occupied in the disk. The field is the base size of the file or directory before replication.

```
$ hadoop fs -du  /hdfs-file-path
or
$ hdfs dfs -du  /hdfs-file-path
```

dus – Directory/file of the total size

Dus command is used to will give the total size of directory/file.

```
$ hadoop fs -dus  /hdfs-directory
or
$ hdfs dfs -dus  /hdfs-directory
```

get – Copy the File from HDFS to Local

Get command is used to store filess from HDFS to the local file. HDFS file gets the local machine.

```
$ hadoop fs -get /local-file-path /hdfs-file-path
or
$ hdfs dfs -get /local-file-path /hdfs-file-path
```

getmerge – Merge Multiple Files in an HDFS

If you have multiple files in an HDFS, use -getmerge option command. All these multiple files merged into one single file and downloads to local file system.

```
$ hadoop fs -getmerge [-nl] /source /local-destination
or
$ hdfs dfs -getmerge [-nl] /source /local-destination
```

count – Number of Directory

The count command is used to count a number of directories, a number of files, and file size on HDFS.

```
$ hadoop fs -count /hdfs-file-path
or
$ hdfs dfs -count /hdfs-file-path
```

mv – Moves Files from Source to Destination

MV (move) command is used to move files from one location to another location in HDFS. Move command allows multiple sources as well in which case the destination needs to be a director.

```
$ hadoop fs -mv /local-file-path /hdfs-file-path
or
$ hdfs dfs -mv /local-file-path /hdfs-file-path
```

moveFromLocal – Move file / Folder from Local disk to HDFS

Similar to the put command, `moveFromLocal` moves the file or source from the local file path to the destination in the HDFS file path. After this command, you will not find the file on the local file system.

```
$ hadoop fs -moveFromLocal /local-file-path /hdfs-file-path
or
$ hdfs dfs -moveFromLocal /local-file-path /hdfs-file-path
```

moveToLocal – Move a File to HDFS from Local

Similar to the get command, moveToLocal moves the file or source from the HDFS file path to the destination in the local file path.

```
```

```
$ hadoop fs -moveToLocal /hdfs-file-path /local-file-path
or
$ hdfs dfs -moveToLocal /hdfs-file-path /local-file-path
```

Cp – Copy Files from Source to Destination

Copy File-one location to another location in HDFS. Copy files from source to destination, Copy command allows multiple sources as well in which case the destination must be a directory.

```
$ hadoop fs -cp /local-file-path /hdfs-file-path
or
$ hdfs dfs -cp /local-file-path /hdfs-file-path
```

setrep – Changes the Replication Factor of a File

This HDFS command is used to change the replication factor of a file. If the path is a directory then the command recursively changes the replication factor of all files under the directory tree rooted at the path.

```
$ hadoop fs -setrep /number /file-name
or
$ hdfs dfs -setrep /number /file-name
```

tail – Displays Last Kilobyte of the File

Tail command is used to Display last kilobyte of the file to stdout.

```
$ hadoop fs -tail /hdfs-file-path
or
$ hdfs dfs -tail /hdfs-file-path
```

touch – Create and Modify Timestamps of a File

It is used to create a file without any content. The file created using the touch command is empty. updates the access and modification times of the file specified by the URI to the current time, the file does not exist then a zero-length file is created at URI with the current time as the timestamp of that URI.

```
$ hadoop fs -touch /hdfs-file-path
or
```

```
$ hdfs dfs -touch /hdfs-file-path
```

touchz – Create a File of zero Length

Create a new file on HDFS with size 0 bytes. create a file of zero length, an error is returned if the file exists with non-zero length.

```
$ hadoop fs -touchz /hdfs-file-path
or
$ hdfs dfs -touchz /hdfs-file-path
```

appendToFile – Appends the Content to the File

Appends the content to the file which is present on HDFS. Append single source. or multiple sources from the local file system to the destination file system. this command appends the contents of all the given local files to the provided destination file on the HDFS filesystem.

```
$ hadoop fs -appendToFile /hdfs-file-path
or
$ hdfs dfs -appendToFile /hdfs-file-path
```

copyFromLocal – Copy File from Local file System

Copying file from a local file to HDFS file system. Similar to the fs -put command and copyFromLocal command both are Store files from local disk to HDFS. Except that the source is restricted to a local file reference.

```
$ hadoop fs -copyToLocal /hdfs-file-path /local-file-path
or
$ hdfs dfs -copyToLocal  /hdfs-file-path /local-file-path
```

copyToLocal – Copy Files from HDFS to Local file System

Copying files from HDFS file to local file system. Similar to the fs -get command and copyToLocal command both are Store files from hdfs to local files. Except that the destination is restricted to a local file reference.

```
$ hadoop fs -copyToLocal /hdfs-file-path /local-file-path
or
$ hdfs dfs -copyToLocal  /hdfs-file-path /local-file-path
```

usage – Return the Help for Individual Command

Usage command is used to Provide you help for indidual commands.

```
$ hadoop fs -usage mkdir
or
$ hdfs dfs -usage mkdir
```

checksum -Returns the Checksum Information of a File

The checksum command is used to Returns the Checksum Information of a File. Returns the checksum information of a file.

```
$ hadoop fs -checksum [-v] URI
or
$ hdfs dfs -checksum [-v] URI
```

chgrp – Change Group Association of Files

chgrg command is used to change the group of a file or a path. The user must be the owner of files, or else a super-user.

```
$ hadoop fs -chgrp [-R] groupname
or
$ hdfs dfs -chgrp [-R] groupname
```

chmod – Change the Permissions of a File

This command is used to change the permissions of a file. With -R Used to modify the files recursively and it is the only option that is being supported currently.

```
$ hadoop fs -chmod [-R] hdfs-file-path
or
$ hdfs dfs -chmod [-R] hdfs-file-path
```

chown – Change the Owner and Group of a File

Chown command is used to change the owner and group of a file. This command is similar to the shell's chown command with a few exceptions.

```
$ hadoop fs -chown [-R] [owner][:[group]] hdfs-file-path
or
```

```
$ hdfs dfs -chown [-R] [owner][:[group]] hdfs-file-path
```

df – Displays free Space

Df is the Displays free space. This command is used to show the capacity, free and used space available on the HDFS filesystem. Used to format the sizes of the files in a human-readable manner rather than the number of bytes.

```
$ hadoop fs -df /user/hadoop/dir1
or
$ hdfs dfs -df /user/hadoop/dir1
```

head – Displays first Kilobyte of the File

Head command is use to Displays first kilobyte of the file to stdout.

```
$ hadoop fs -head /hdfs-file-path
or
$ hdfs dfs -head /hdfs-file-path
```

createSnapshots – Create Snapshottable Directory

This operation requires owner privilege of the snapshot table directory. The path of the snapshot table directory, snapshot name is The snapshot name a default name is generated using a timestamp.

```
$ hadoop fs -createSnapshot /path /snapshotName
or
$ hdfs dfs -createSnapshot /path /snapshotName
```

deleteSnapshots – Delect Snapshottable Directory

This operation requires owner privilege of the snapshot table directory. The path of the snapshot table directory, snapshot name is The snapshot name.

```
$ hadoop fs -deleteSnapshot /path /snapshotName
or
$ hdfs dfs -deleteSnapshot /path /snapshotName
```

renameSnapshots – Rename a Snapshot

This operation requires owner privilege of the snapshottable directory.

```
$ hadoop fs -renameSnapshot /path /oldName /newName
```

```
or
$ hdfs dfs  -renameSnapshot /path /oldName /newName
```

expunge – Create New Checkpoint

This command is used to empty the trash available in an HDFS system. Permanently delete files in checkpoints older than the retention threshold from the trash directory.

```
$ hadoop fs –expunge -immediate -fs  /hdfs-file-path
or
$ hdfs dfs –expunge -immediate -fs  /hdfs-file-path
```

Stat – File/Directory Print Statistics

This command is used to print the statistics about the file/directory in the specified format. Print statistics about the file/directory at in the specified format.

```
$ hadoop fs -stat /format
or
$ hdfs dfs -stat /format
```

Truncate – Specified File Pattern and Length

Truncate all files that match the specified file pattern to the specified length.

```
$ hadoop fs -truncate [-w] /length /hdfs-file-path
or
$ hdfs dfs -truncate [-w] /length /hdfs-file-path
```

Find – Find File Size in HDFS

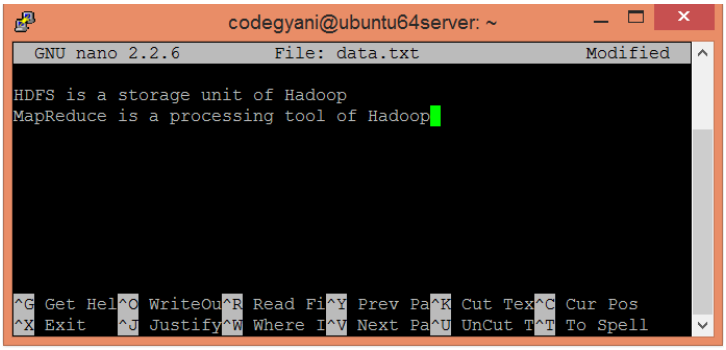
In Hadoop, `hdfs dfs -find` or `hadoop fs -find` commands are used to get the size of a single file or size for all files specified in an expression or in a directory. By default, it points to the current directory when the path is not specified.

```
$hadoop fs -find / -name test -print
or
$hdfs dfs -find / -name test -print
```

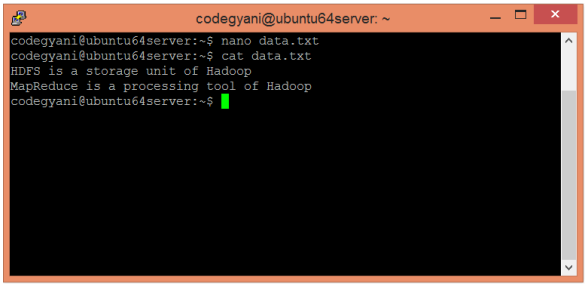
b) Run a basic word count Map Reduce program to understand Map Reduce

Steps to execute MapReduce word count

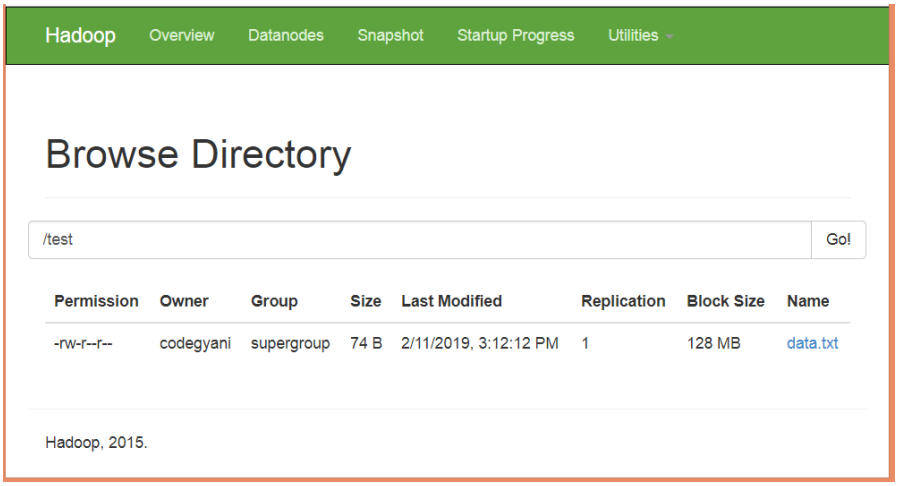
- Create a text file in your local machine and write some text into it.
  $ nano data.txt



- Check the text written in the data.txt file.
  $ cat data.txt



- create a directory in HDFS, where to kept text file.
  $ hdfs dfs -mkdir /test
- Upload the data.txt file on HDFS in the specific directory.
  $ hdfs dfs -put /home/codegyani/data.txt /test



Write the MapReduce program using eclipse.

WC_Mapper.java

```
package com.javatpoint;
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
public class WC_Mapper extends MapReduceBase implements Mapper<LongWritable,Text,
Text,IntWritable>{
   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();
   public void map(LongWritable key, Text value,OutputCollector<Text,IntWritable> output,
```

```
        Reporter reporter) throws IOException{
      String line = value.toString();
      StringTokenizer  tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()){
         word.set(tokenizer.nextToken());
         output.collect(word, one);
      }
   }

}
```

WC_Reducer.java

```
package com.javatpoint;
   import java.io.IOException;
   import java.util.Iterator;
   import org.apache.hadoop.io.IntWritable;
   import org.apache.hadoop.io.Text;
   import org.apache.hadoop.mapred.MapReduceBase;
   import org.apache.hadoop.mapred.OutputCollector;
   import org.apache.hadoop.mapred.Reducer;
   import org.apache.hadoop.mapred.Reporter;

   public class WC_Reducer  extends MapReduceBase implements Reducer<Text,IntWritable,
Text,IntWritable> {
   public void reduce(Text key, Iterator<IntWritable> values,OutputCollector<Text,IntWrita
ble> output,
    Reporter reporter) throws IOException {
   int sum=0;
   while (values.hasNext()) {
```

```
sum+=values.next().get();
}
output.collect(key,new IntWritable(sum));
}
}
```

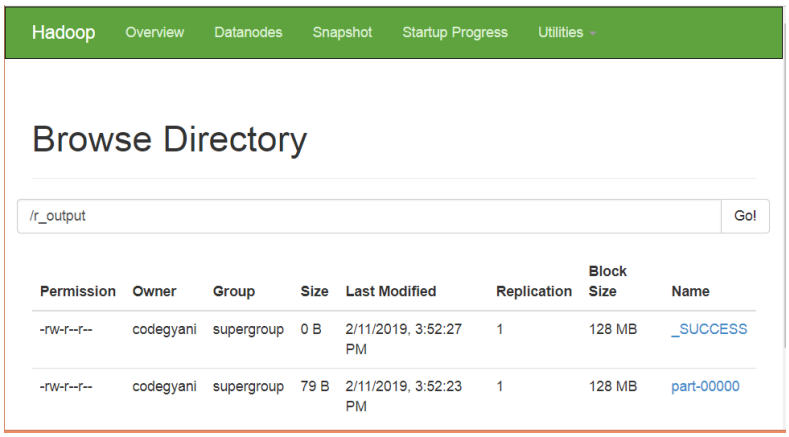
WC_Runner.java

```
package com.javatpoint;
import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.TextOutputFormat;
public class WC_Runner {
   public static void main(String[] args) throws IOException{
      JobConf conf = new JobConf(WC_Runner.class);
      conf.setJobName("WordCount");
      conf.setOutputKeyClass(Text.class);
      conf.setOutputValueClass(IntWritable.class);
      conf.setMapperClass(WC_Mapper.class);
      conf.setCombinerClass(WC_Reducer.class);
      conf.setReducerClass(WC_Reducer.class);
```

```
        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);
        FileInputFormat.setInputPaths(conf,new Path(args[0]));
        FileOutputFormat.setOutputPath(conf,new Path(args[1]));
        JobClient.runJob(conf);
    }
}
```

Download the source code.

- Create the jar file of this program and name it **countworddemo.jar.**
- Run the jar file
  hadoop jar /home/codegyani/wordcountdemo.jar com.javatpoint.WC_Runner /test/data.txt /r_output
- The output is stored in /r_output/part-00000



- Now execute the command to see the output.
  hdfs dfs -cat /r_output/part-00000


codegyani@ubuntu64server: ~
codegyani@ubuntu64server:~$ hdfs dfs -cat /r_output/part-00000
HDFS 1
Hadoop 2
MapReduce 1
a 2
is 2
of 2
processing 1
storage 1
tool 1
unit 1
codegyani@ubuntu64server:~$

TASK 2

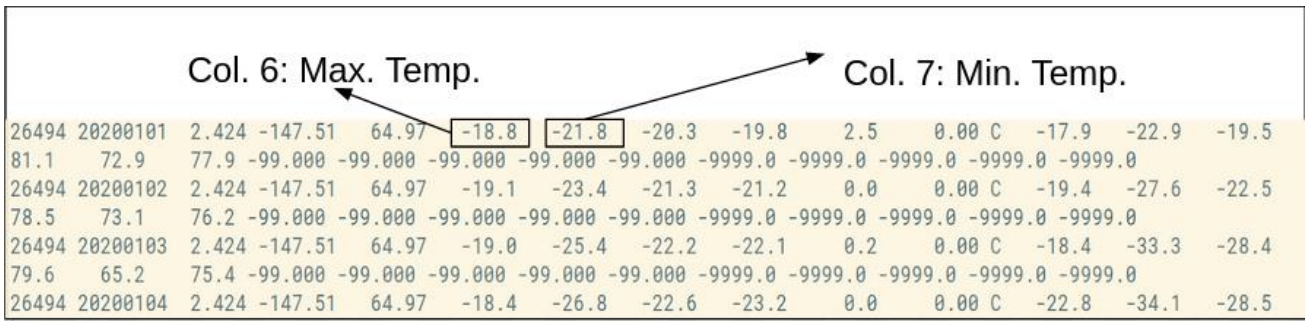
Aim: Write a Map Reduce program that mines weather data

Step 1:

We can download the dataset from this Link, For various cities in different years. choose the year of your choice and select any one of the data text-file for analyzing. In my case, I have selected *CRND0103-2020-AK_Fairbanks_11_NE.txt* dataset for analysis of hot and cold days in Fairbanks, Alaska.
We can get information about data from *README.txt* file available on the NCEI website.

Step 2:

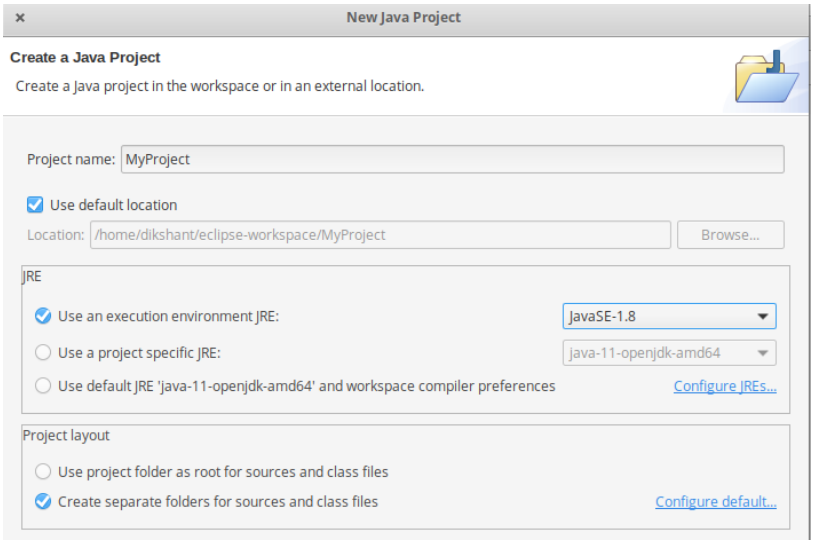
Below is the example of our dataset where column 6 and column 7 is showing Maximum and Minimum temperature, respectively.



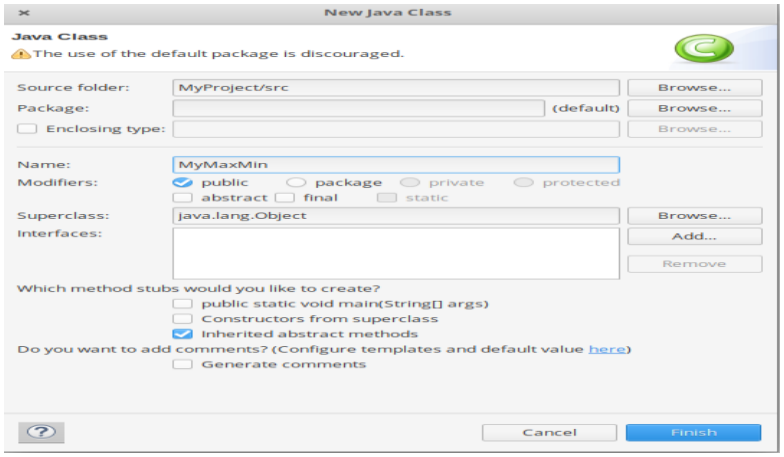
Step 3:

Make a project in Eclipse with below steps:

- First Open **Eclipse** -> then select **File -> New -> Java Project** ->Name it **MyProject** -> then select **use an execution environment** -> choose **JavaSE-1.8** then **next** -> **Finish**.



- In this Project Create Java class with name **MyMaxMin** -> then click **Finish**



- Copy the below source code to this **MyMaxMin** java class

```
// importing Libraries
import java.io.IOException;
import java.util.Iterator;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
```

```
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.conf.Configuration;

public class MyMaxMin {


        // Mapper

        /*MaxTemperatureMapper class is static
        * and extends Mapper abstract class
        * having four Hadoop generics type
        * LongWritable, Text, Text, Text.
        */


        public static class MaxTemperatureMapper extends
                        Mapper<LongWritable, Text, Text, Text> {

                /**
                * @method map
                * This method takes the input as a text data type.
                * Now leaving the first five tokens, it takes
                * 6th token is taken as temp_max and
                * 7th token is taken as temp_min. Now
                * temp_max > 30 and temp_min < 15 are
                * passed to the reducer.
                */

        // the data in our data set with
        // this value is inconsistent data
        public static final int MISSING = 9999;
```

```
@Override
        public void map(LongWritable arg0, Text Value, Context context)
                        throws IOException, InterruptedException {

                // Convert the single row(Record) to
                // String and store it in String
                // variable name line

                String line = Value.toString();

                        // Check for the empty line
                        if (!(line.length() == 0)) {

                                // from character 6 to 14 we have
                                // the date in our dataset
                                String date = line.substring(6, 14);

                                // similarly we have taken the maximum
                                // temperature from 39 to 45 characters
                                float temp_Max = Float.parseFloat(line.substring(39,
45).trim());

                                // similarly we have taken the minimum
                                // temperature from 47 to 53 characters

                                float temp_Min = Float.parseFloat(line.substring(47,
53).trim());

                                // if maximum temperature is
                                // greater than 30, it is a hot day
                                if (temp_Max > 30.0) {

                                        // Hot day
                                        context.write(new Text("The Day is Hot Day :" +
date),
                                                                        new
Text(String.valueOf(temp_Max)));
                                }
```

```
                                        // if the minimum temperature is
                                        // less than 15, it is a cold day
                                        if (temp_Min < 15) {

                                                // Cold day
                                                context.write(new Text("The Day is Cold Day :" +
date),
                                                                new
Text(String.valueOf(temp_Min)));
                                        }
                                }
                        }

                }

// Reducer

                /*MaxTemperatureReducer class is static
                and extends Reducer abstract class
                having four Hadoop generics type
                Text, Text, Text, Text.
                */

                public static class MaxTemperatureReducer extends
                                Reducer<Text, Text, Text, Text> {

                        /**
                        * @method reduce
                        * This method takes the input as key and
                        * list of values pair from the mapper,
                        * it does aggregation based on keys and
                        * produces the final context.
                        */

                        public void reduce(Text Key, Iterator<Text> Values, Context context)
                                        throws IOException, InterruptedException {
```

```
                // putting all the values in
                // temperature variable of type String
                String temperature = Values.next().toString();
                context.write(Key, new Text(temperature));
        }

}



/**
* @method main
* This method is used for setting
* all the configuration properties.
* It acts as a driver for map-reduce
* code.
*/

public static void main(String[] args) throws Exception {

        // reads the default configuration of the
        // cluster from the configuration XML files
        Configuration conf = new Configuration();

        // Initializing the job with the
        // default configuration of the cluster
        Job job = new Job(conf, "weather example");

        // Assigning the driver class name
        job.setJarByClass(MyMaxMin.class);

        // Key type coming out of mapper
        job.setMapOutputKeyClass(Text.class);

        // value type coming out of mapper
        job.setMapOutputValueClass(Text.class);
```

```
        // Defining the mapper class name
        job.setMapperClass(MaxTemperatureMapper.class);

        // Defining the reducer class name
        job.setReducerClass(MaxTemperatureReducer.class);

        // Defining input Format class which is
        // responsible to parse the dataset
        // into a key value pair
        job.setInputFormatClass(TextInputFormat.class);

        // Defining output Format class which is
        // responsible to parse the dataset
        // into a key value pair
        job.setOutputFormatClass(TextOutputFormat.class);

        // setting the second argument
        // as a path in a path variable
        Path OutputPath = new Path(args[1]);

        // Configuring the input path
        // from the filesystem into the job
        FileInputFormat.addInputPath(job, new Path(args[0]));

        // Configuring the output path from
        // the filesystem into the job
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // deleting the context path automatically
        // from hdfs so that we don't have
        // to delete it explicitly
        OutputPath.getFileSystem(conf).delete(OutputPath);

        // exiting the job only if the
        // flag value becomes false
        System.exit(job.waitForCompletion(true) ? 0 : 1);

}
```

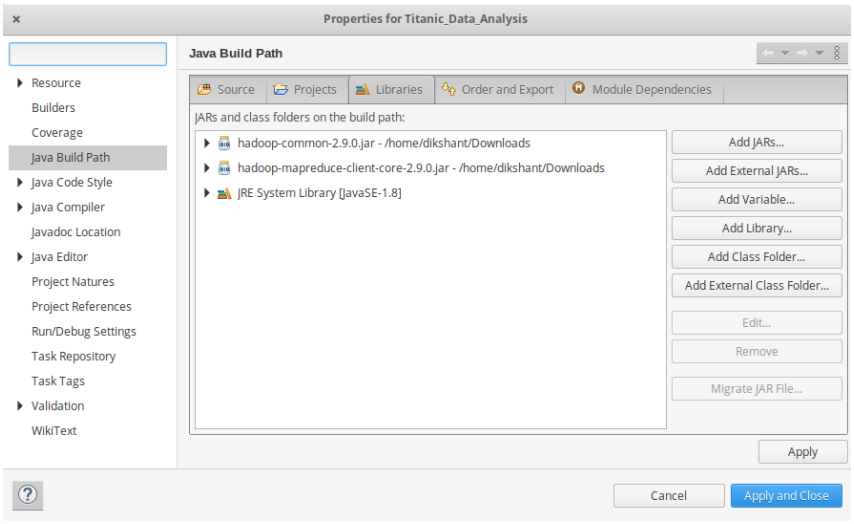
}

Now we need to add external jar for the packages that we have import. Download the jar package Hadoop Common and Hadoop MapReduce Core according to your Hadoop version.

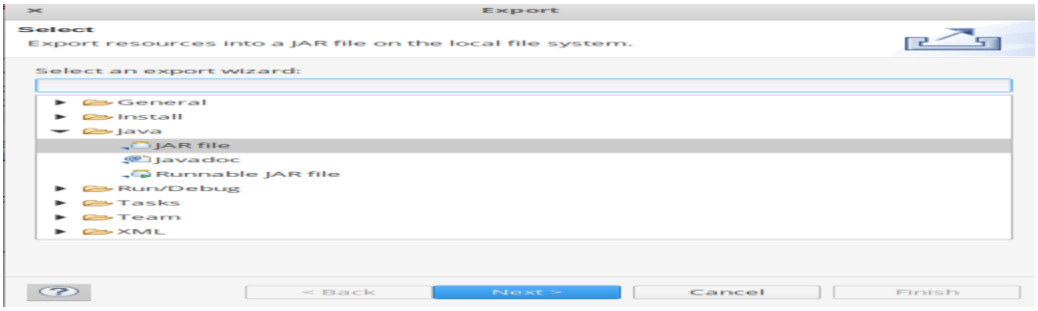
You can check Hadoop Version:

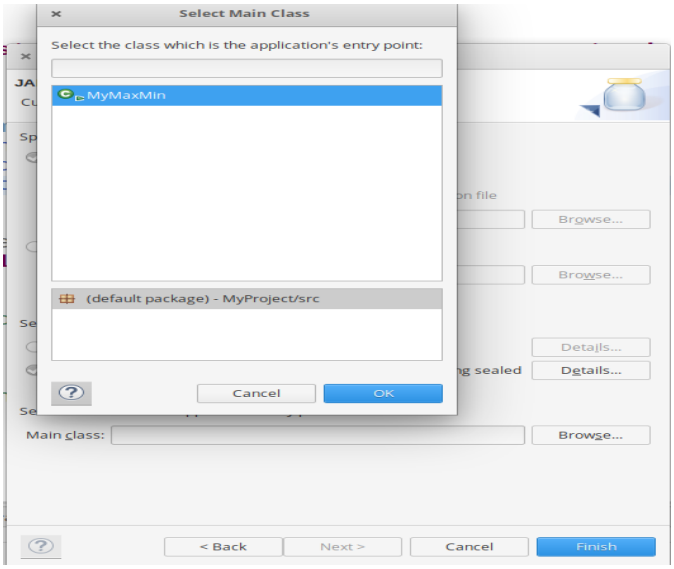
```
hadoop version
```

- Now we add these external jars to our **MyProject**. Right Click on **MyProject** -> then select **Build Path**-> Click on **Configure Build Path** and select **Add External jars….** and add jars from it's download location then click -> **Apply and Close**.



- Now export the project as jar file. Right-click on **MyProject** choose **Export..** and go to **Java -> JAR file** click -> **Next** and choose your export destination then click -> **Next**.
  choose Main Class as **MyMaxMin** by clicking -> **Browse** and then click -> **Finish** -> **Ok**.





Step 4:

Start our Hadoop Daemons

start-dfs.sh

start-yarn.sh

Step 5:

Move your dataset to the Hadoop HDFS.
Syntax:

hdfs dfs -put /file_path /destination

In below command / shows the root directory of our HDFS.

hdfs dfs -put /home/dikshant/Downloads/CRND0103-2020-AK_Fairbanks_11_NE.txt /

Check the file sent to our HDFS.

hdfs dfs -ls /

Step 6:

Now Run your Jar File with below command and produce the output in **MyOutput** File.
Syntax:
hadoop jar /jar_file_location /dataset_location_in_HDFS /output-file_name

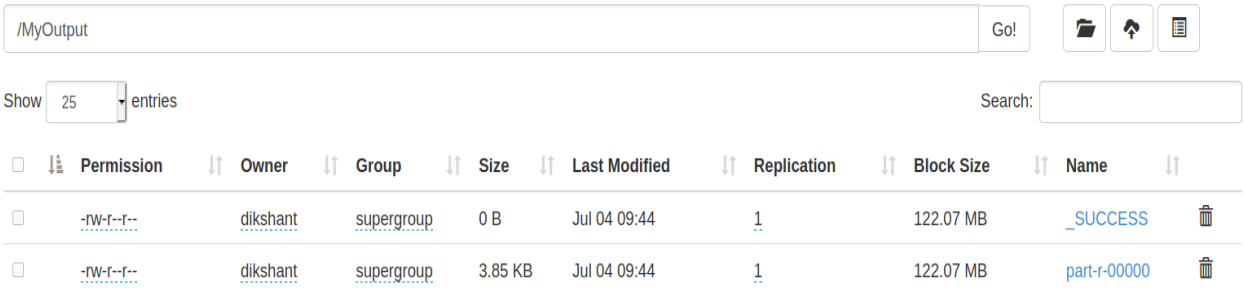
Command:

```
hadoop jar /home/dikshant/Documents/Project.jar /CRND0103-2020-AK_Fairbanks_11_NE.txt /MyOutput
```

Step 7:

Now Move to *localhost:50070/*, under utilities select *Browse the file system* and download **part-r-00000** in **/MyOutput** directory to see result.

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	dikshant	supergroup	38.78 KB	Jul 04 09:39	1	122.07 MB	CRND0103-2020-AK_Fairbanks_11_NE.txt
drwxrwxr-x+	dikshant	supergroup	0 B	Jun 23 14:23	0	0 B	Hadoop_File
drwxr-xr-x	dikshant	supergroup	0 B	Jul 04 09:44	0	0 B	MyOutput
drwxrwxrwx	dikshant	supergroup	0 B	Jun 14 21:43	0	0 B	tmp
drwxr-xr-x	dikshant	supergroup	0 B	Jun 14 21:43	0	0 B	user



Step 8:

See the result in the Downloaded File.

```
1 The Day is Cold Day :20200101   -21.8
2 The Day is Cold Day :20200102   -23.4
3 The Day is Cold Day :20200103   -25.4
4 The Day is Cold Day :20200104   -26.8
5 The Day is Cold Day :20200105   -28.8
6 The Day is Cold Day :20200106   -30.0
7 The Day is Cold Day :20200107   -31.4
8 The Day is Cold Day :20200108   -33.6
9 The Day is Cold Day :20200109   -26.6
10 The Day is Cold Day :20200110   -24.3
```

In the above image, you can see the top 10 results showing the cold days. The second column is a day in yyyy/mm/dd format. For Example, **20200101** means
year = 2020

month = 01

Date = 01

TASK 3: Implement matrix multiplication with Hadoop Map Reduce.

Mapper Logic :

```
import java.io.IOException;

import org.apache.hadoop.conf.*;

import org.apache.hadoop.io.*;

import org.apache.hadoop.mapreduce.*;

public class MatrixMapper extends Mapper<LongWritable, Text, Text, Text>

{

public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException

{

Configuration conf = context.getConfiguration();

int m = Integer.parseInt(conf.get("m"));

int p = Integer.parseInt(conf.get("p"));

String line = value.toString();

String[] indicesAndValue = line.split(",");

Text outputKey = new Text();

Text outputValue = new Text();

if (indicesAndValue[0].equals("M"))
```

```
{
        for (int k = 0; k < p; k++)
{
outputKey.set(indicesAndValue[1] + "," + k);
outputValue.set("M," + indicesAndValue[2] + "," + indicesAndValue[3]);
context.write(outputKey, outputValue);
}
}
else
{
for (int i = 0; i < m; i++)
{
outputKey.set(i + "," + indicesAndValue[2]);
outputValue.set("N," + indicesAndValue[1] + "," + indicesAndValue[3]);
context.write(outputKey, outputValue);
}
}
}
}
```

Reducer Logic :

```
import java.io.IOException;

import java.util.*;

import org.apache.hadoop.io.*;

import org.apache.hadoop.mapreduce.*;

public class MatrixReducer extends Reducer<Text, Text, Text, Text>

{

public void reduce(Text key, Iterable<Text> values, Context context) throws IOException,
InterruptedException

{

String[] value;

HashMap<Integer, Float> hashA = new HashMap<Integer, Float>();

HashMap<Integer, Float> hashB = new HashMap<Integer, Float>();

for (Text val : values)

{

value = val.toString().split(",");

if (value[0].equals("M"))

{

hashA.put(Integer.parseInt(value[1]), Float.parseFloat(value[2]));

}
```

```
else
{
hashB.put(Integer.parseInt(value[1]), Float.parseFloat(value[2]));
}
}
int n = Integer.parseInt(context.getConfiguration().get("n")); float result = 0.0f;
float a_ij; float b_jk;
for (int j = 0; j < n; j++)
{
a_ij = hashA.containsKey(j) ? hashA.get(j) : 0.0f;
b_jk = hashB.containsKey(j) ? hashB.get(j) : 0.0f;
result += a_ij * b_jk;
}
if (result != 0.0f)
{
context.write(null, new Text(key.toString() + "," + Float.toString(result)));
}
}
}
```

Driver Logic :

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class MatrixDriver
{
        public static void main(String[] args) throws Exception
        {
                Configuration conf = new Configuration();
                // M is an m-by-n matrix; N is an n-by-p matrix.
                conf.set("m", "2");
```

```
        conf.set("n", "2");
        conf.set("p", "2");
        Job job = Job.getInstance(conf, "MatrixMultiplication");
        job.setJarByClass(MatrixDriver.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        job.setMapperClass(MatrixMapper.class);
        job.setReducerClass(MatrixReducer.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.submit();
}
}
```

Output:

TASK-4

Aim: Working with files in Hadoop file system: Reading, Writing and Copying

Read & Write files in HDFS

Hadoop provides mainly two classes FSDataInputStream for reading a file from HDFS and FSDataOutputStream for writing a file to HDFS.

Development Environment
Hadoop: 3.1.1

Java: Oracle JDK 1.8

IDE: IntelliJ Idea 2018.3

Initialize Configuration
First step in communication with HDFS is to initialize Configuration class and set fs.defaultFS property. Refer below code snippet.

```
Configuration configuration = new Configuration();
configuration.set("fs.defaultFS", "hdfs://localhost:9000");
```

Create Directory in HDFS

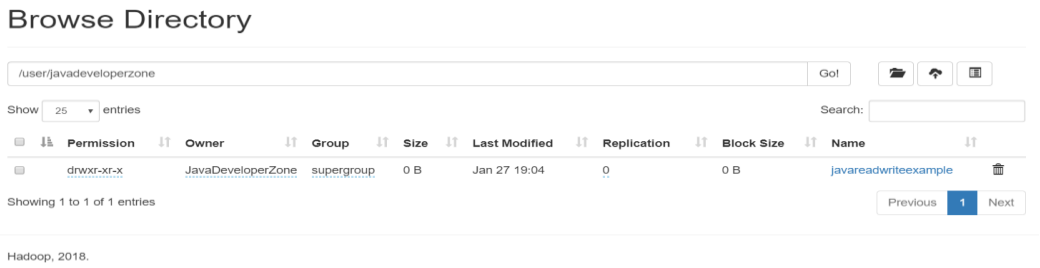
Hadoop FileSystem class provide all the admin related functionality like create file or directory, delete file etc. mkDirs method is used to create a directory under HDFS.

Program:

```
public static void createDirectory() throws IOException {
        Configuration configuration = new Configuration();
        configuration.set("fs.defaultFS", "hdfs://localhost:9000");
        FileSystem fileSystem = FileSystem.get(configuration);
        String directoryName = "javadeveloperzone/javareadwriteexample";
        Path path = new Path(directoryName);
        fileSystem.mkdirs(path);
    }
```

Output

Go to HDFS web view and everything is running fine you will see a directory javareadwriteexample under /user/javadeveloperzone path.



Read Write HDFS

Write File to HDFS

FSDataOutputStream class used to write data to HDFS file. It also provides various methods like writeUTF, writeInt, WriteChar etc..Here we have wrapped FSDataOutputStream to BufferedWrite class.

Program

```
public static void writeFileToHDFS() throws IOException {
        Configuration configuration = new Configuration();
        configuration.set("fs.defaultFS", "hdfs://localhost:9000");
        FileSystem fileSystem = FileSystem.get(configuration);
        //Create a path
        String fileName = "read_write_hdfs_example.txt";
        Path hdfsWritePath = new Path("/user/javadeveloperzone/javareadwriteexample/" +
fileName);
        FSDataOutputStream fsDataOutputStream = fileSystem.create(hdfsWritePath,true);
        BufferedWriter bufferedWriter = new BufferedWriter(new
OutputStreamWriter(fsDataOutputStream,StandardCharsets.UTF_8));
        bufferedWriter.write("Java API to write data in HDFS");
        bufferedWriter.newLine();
        bufferedWriter.close();
        fileSystem.close();
    }
```

Append Data to File

FileSystem class append method is used to append data to an existing file.

Program:

```
public static void appendToHDFSFile() throws IOException {
      Configuration configuration = new Configuration();
      configuration.set("fs.defaultFS", "hdfs://localhost:9000");
      FileSystem fileSystem = FileSystem.get(configuration);
      //Create a path
      String fileName = "read_write_hdfs_example.txt";
      Path hdfsWritePath = new Path("/user/javadeveloperzone/javareadwriteexample/" +
fileName);
      FSDataOutputStream fsDataOutputStream = fileSystem.append(hdfsWritePath);
      BufferedWriter bufferedWriter = new BufferedWriter(new
OutputStreamWriter(fsDataOutputStream,StandardCharsets.UTF_8));
      bufferedWriter.write("Java API to append data in HDFS file");
      bufferedWriter.newLine();
      bufferedWriter.close();
      fileSystem.close();
   }
```

Read File From HDFS

FSDataInputStream class provide facility to read a file from HDFS.

Program

```
public static void readFileFromHDFS() throws IOException {
      Configuration configuration = new Configuration();
      configuration.set("fs.defaultFS", "hdfs://localhost:9000");
      FileSystem fileSystem = FileSystem.get(configuration);
      //Create a path
      String fileName = "read_write_hdfs_example.txt";
      Path hdfsReadPath = new Path("/user/javadeveloperzone/javareadwriteexample/" +
fileName);
      //Init input stream
```

```
FSDataInputStream inputStream = fileSystem.open(hdfsReadPath);
//Classical input stream usage
String out= IOUtils.toString(inputStream, "UTF-8");
System.out.println(out);
/*BufferedReader bufferedReader = new BufferedReader(
      new InputStreamReader(inputStream, StandardCharsets.UTF_8));
String line = null;
while ((line=bufferedReader.readLine())!=null){
   System.out.println(line);
}*/
inputStream.close();
fileSystem.close();
```

Output

Java API to write data in HDFS
Java API to append data in HDFS file

Copy file from local disk to hdfs using java

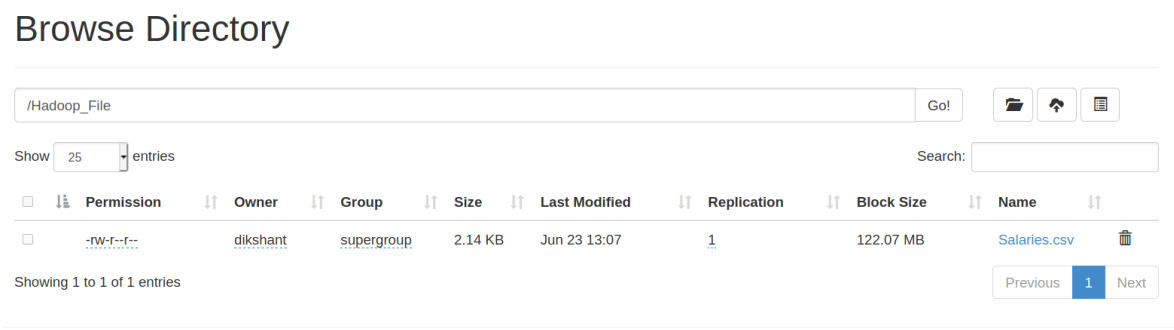
Create a project in eclipse or netbeans or any editor you like and add hadoop-core.jar and create a class named PutToHdfs and put this code

Program

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.URI;
import java.net.URISyntaxException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.hdfs.DistributedFileSystem;


public class PutToHdfs {
```

this contains two variation of file copying to hdfs your accordingly as your requirement, explanation given in comment.

```
    */
   public static void main(String[] args) throws IOException, URISyntaxException {
      // TODO code application logic here
      //InetSocketAddress add = new InetSocketAddress("192.168.20.12", 9000);   //-------> Use this if you are using DistributedFileSystem Class(For hadoop configured as distributed)
      URI url=new URI("hdfs://192.168.0.1:9000"); //-------> (url where hdfs located-for detail look hadoop configuration )Use this if you are using FileSystem Class(For hadoop configured on a single system)
      Configuration conf = new Configuration();
      //DistributedFileSystem ffs = new DistributedFileSystem(add, conf);
       FileSystem file1= FileSystem.get(url, conf);
      Path src = new Path("/sourcepath/filename.text");
      Path dst = new Path("/destinationhdfsfolder/");
      //fs.copyFromLocalFile(b1, b2, src, dst);
      file1.copyFromLocalFile(src, dst);
   }
}
```



TASK-5

Aim : Write Pig Latin scripts sort, group, join, project, and filter your data

Apache Pig create a simpler procedural language abstraction over Map Reduce to expose a more Structured Query Language (SQL)-like interface for Hadoop applications called Apache Pig Latin, So instead of writing a separate Map Reduce application, you can write a single script in Apache Pig Latin that is automatically parallelized and distributed across a cluster. In simple words, Pig Latin, is a sequence of simple statements taking an input and producing an output. The input and output data are composed of bags, maps, tuples and scalar.

Apache Pig Execution Modes:

Apache Pig has two execution modes:

- ***Local Mode***

In 'Local Mode', the source data would be picked from the local directory in your computer system. The MapReduce mode can be specified using 'pig –x local' command.

```
cloudera@cloudera-vm:~$ pig -x local
2013-11-21 23:08:28,088 [main] INFO  org.apache.pig.Main - Logging error messages to: /home/cloudera/pig_1385104108087.log
2013-11-21 23:08:28,208 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop fil
e system at: file:///
grunt>
```

- ***MapReduce Mode:***

To run Pig in MapReduce mode, you need access to Hadoop cluster and HDFS installation. The MapReduce mode can be specified using the 'pig' command.

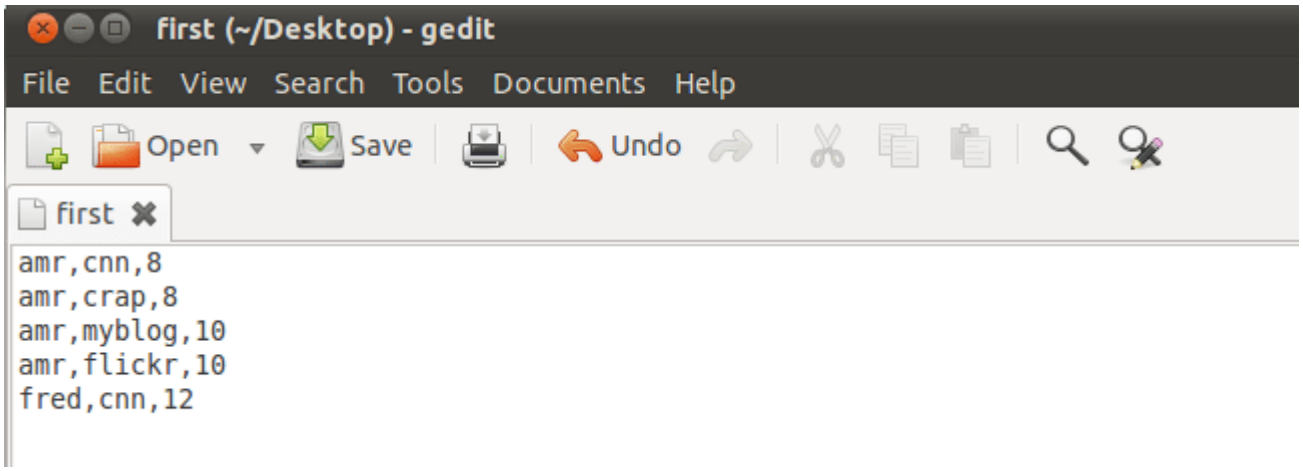
```
cloudera@cloudera-vm:~$ pig
2013-11-21 22:54:24,964 [main] INFO  org.apache.pig.Main - Logging error messages to: /home/cloudera/pig_1385103264962.log
2013-11-21 22:54:25,154 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop fil
e system at: hdfs://localhost:8020
2013-11-21 22:54:25,405 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to map-reduce
 job tracker at: localhost:8021
grunt>
```

Apache Pig Operators:

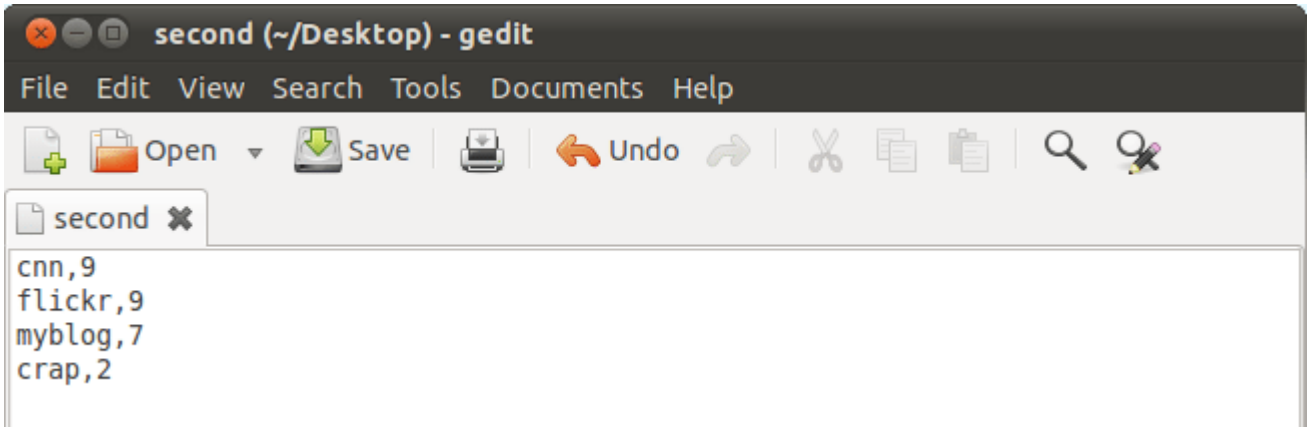
The Apache Pig Operators is a high-level procedural language for querying large data sets using Hadoop and the Map Reduce Platform. A Pig Latin statement is an operator that takes a relation as input and produces another relation as output. These operators are the main tools for Pig Latin provides to operate on the data. They allow you to transform it by sorting, grouping, joining, projecting, and filtering.

Let's create two files to run the commands:

We have two files with name 'first' and 'second.' The first file contain three fields: user, url & id.



The second file contain two fields: url & rating. These two files are CSV files.



The Apache Pig operators can be classified as: *Relational and Diagnostic.*

Relational Operators:

Relational operators are the main tools Pig Latin provides to operate on the data. It allows you to transform the data by sorting, grouping, joining, projecting and filtering. This section covers the basic relational operators.

LOAD:

LOAD operator is used to load data from the file system or HDFS storage into a Pig relation.

In this example, the Load operator loads data from file 'first' to form relation 'loading1'. The field names are user, url, id.

```
grunt> loading1 = load '/first' using PigStorage(',') as(user:chararray,url:chararray,id:int);
grunt>
```

```
grunt> loading2 = load '/second' using PigStorage(',') as(url:chararray,rating:int);
grunt>
```

FOREACH:

This operator generates data transformations based on columns of data. It is used to add or remove fields from a relation. Use FOREACH-GENERATE operation to work with columns of data.

```
grunt> for_each = foreach loading1 generate url,id;
grunt> dump for_each;
```

FOREACH Result:

```
(cnn,8)
(crap,8)
(myblog,10)
(flickr,10)
(cnn,12)
grunt>
```

FILTER:

This operator selects tuples from a relation based on a condition.

In this example, we are filtering the record from 'loading1' when the condition 'id' is greater than 8.

```
grunt> filter_command = filter loading1 by id>8;
grunt> dump filter_command;
```

FILTER Result:

```
(amr,myblog,10)
(amr,flickr,10)
(fred,cnn,12)
grunt>
```

JOIN:

JOIN operator is used to perform an inner, equijoin join of two or more relations based on common field values. The JOIN operator always performs an inner join. Inner joins ignore null keys, so it makes sense to filter them out before the join.

In this example, join the two relations based on the column 'url' from 'loading1' and 'loading2'.

```
grunt> join_command = join loading1 by url,loading2 by url;
grunt> dump join_command;
```

JOIN Result:

```
(amr,cnn,8,cnn,9)
(fred,cnn,12,cnn,9)
(amr,crap,8,crap,2)
(amr,flickr,10,flickr,9)
(amr,myblog,10,myblog,7)
grunt>
```

ORDER BY:

Order By is used to sort a relation based on one or more fields. You can do sorting in ascending or descending order using ASC and DESC keywords.

In below example, we are sorting data in loading2 in ascending order on ratings field.

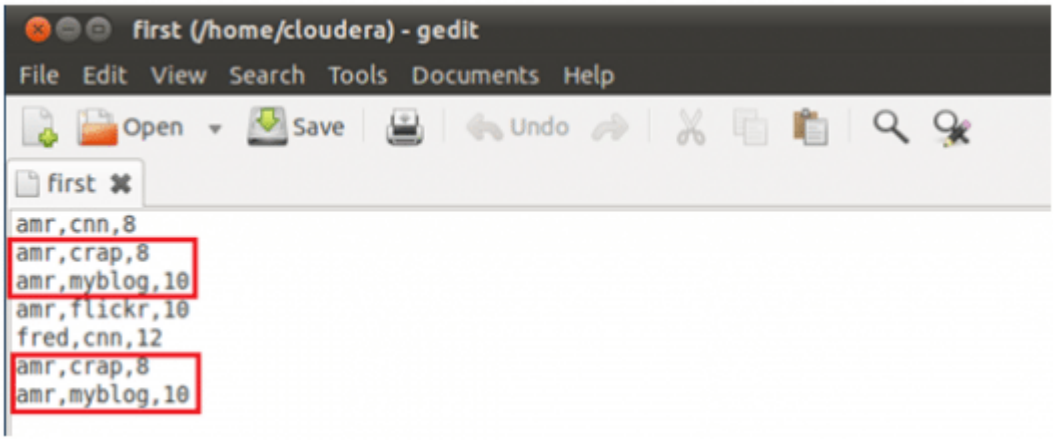
```
grunt> loading4 = ORDER loading2 by rating ASC;
grunt> dump loading4;
```

ORDER BY Result*:*

```
(crap,2)
(myblog,7)
(cnn,9)
(flickr,9)
grunt>
```

DISTINCT:

Distinct removes duplicate tuples in a relation.Lets take an input file as below, which has **amr,crap,8** and **amr,myblog,10** twice in the file. When we apply distinct on the data in this file, duplicate entries are removed.




```
grunt> loading1 = load '/first' using PigStorage(',') as (user:chararray,url:chararray,id:int);
grunt> loading3 = DISTINCT loading1;
grunt> dump loading3;
```

DISTINCT Result:

```
(amr,cnn,8)
(amr,crap,8)
(amr,flickr,10)
(amr,myblog,10)
(fred,cnn,12)
grunt>
```

STORE:

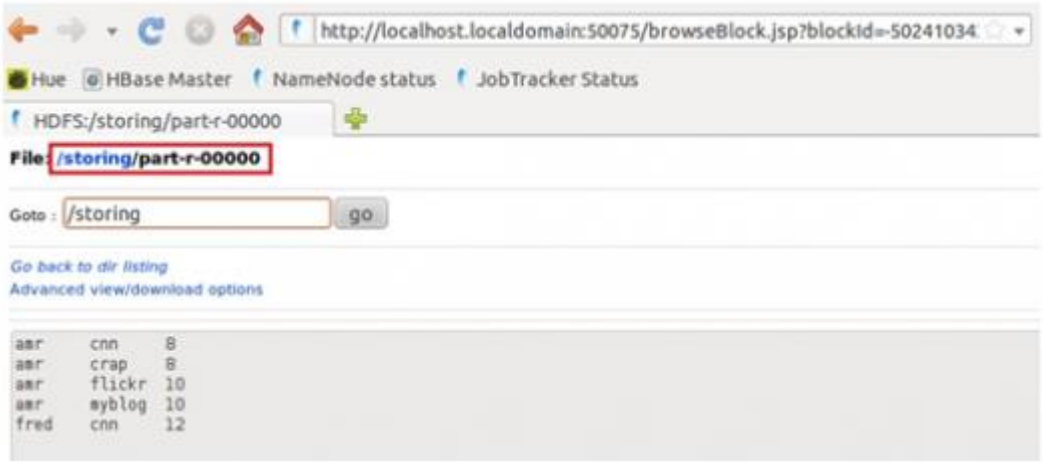
Store is used to save results to the file system.

Here we are saving **loading3** data into a file named **storing** on HDFS.

```
grunt> store loading3 into '/storing';
```

STORE Result:

```
Input(s):
Successfully read 7 records (426 bytes) from: "/first"

Output(s):
Successfully stored 5 records (61 bytes) in: "/storing"
```



GROUP:

The GROUP operator groups together the tuples with the same group key (key field). The key field will be a tuple if the group key has more than one field, otherwise it will be the same type as that of the group key. The result of a GROUP operation is a relation that includes one tuple per group.

In this example, group th

```
grunt> group_command = group loading1 by url;
grunt> dump group_command;
```

e relation 'loading1' by column url.

GROUP Result:

```
(cnn,{(amr,cnn,8),(fred,cnn,12)})
(crap,{(amr,crap,8)})
(flickr,{(amr,flickr,10)})
(myblog,{(amr,myblog,10)})
grunt>
```

COGROUP:

COGROUP is same as GROUP operator. For readability, programmers usually use GROUP when only one relation is involved and COGROUP when multiple relations re involved.

In this example group the ‘loading1’ and ‘loading2’ by url field in both relations.

```
grunt> cogroup_command = cogroup loading1 by url,loading2 by url;
grunt> dump cogroup_command;
```

COGROUP Result:

```
(cnn,{(amr,cnn,8),(fred,cnn,12)},{(cnn,9)})
(crap,{(amr,crap,8)},{(crap,2)})
(flickr,{(amr,flickr,10)},{(flickr,9)})
(myblog,{(amr,myblog,10)},{(myblog,7)})
grunt>
```

CROSS:

The CROSS operator is used to compute the cross product (Cartesian product) of two or more relations.

Applying cross product on loading1 and loading2.

```
grunt> cross_command = cross loading1,loading2;
grunt> dump cross_command;
```

CROSS Result:

```
(fred,cnn,12,crap,2)
(amr,flickr,10,crap,2)
(fred,cnn,12,myblog,7)
(amr,flickr,10,myblog,7)
(amr,flickr,10,cnn,9)
(amr,flickr,10,flickr,9)
(fred,cnn,12,cnn,9)
(fred,cnn,12,flickr,9)
(amr,myblog,10,crap,2)
(amr,myblog,10,myblog,7)
(amr,myblog,10,cnn,9)
(amr,myblog,10,flickr,9)
(amr,cnn,8,crap,2)
(amr,crap,8,crap,2)
(amr,cnn,8,myblog,7)
(amr,crap,8,myblog,7)
(amr,crap,8,cnn,9)
(amr,crap,8,flickr,9)
(amr,cnn,8,cnn,9)
(amr,cnn,8,flickr,9)
grunt>
```

LIMIT:

LIMIT operator is used to limit the number of output tuples. If the specified number of output tuples is equal to or exceeds the number of tuples in the relation, the output will include all tuples in the relation.

```
grunt> limit_command = limit loading1 3;
grunt> dump limit_command;
```

LIMIT Result:

```
(amr,cnn,8)
(amr,crap,8)
(amr,myblog,10)
grunt>
```

SPLIT:

SPLIT operator is used to partition the contents of a relation into two or more relations based on some expression. Depending on the conditions stated in the expression.

Split the loading2 into two relations x and y. x relation created by loading2 contain the fields that the rating is greater than 8 and y relation contain fields that rating is less than or equal to 8.

```
grunt> split loading2 into x if rating>8, y if rating<=8;
grunt> dump x;
```

TASK 6

Run the Pig Latin Scripts to find Word Count and max temp for each and every year

Word Count in Pig Latin

In this Post, we learn how to write word count program using Pig Latin.

Assume we have data in the file like below.

This is a hadoop post

hadoop is a bigdata technology

and we want to generate output for count of each word like below

(a,2)
(is,2)
(This,1)
(class,1)
(hadoop,2)
(bigdata,1)
(technology,1)

Now we will see in steps how to generate the same using Pig latin.

1.Load the data from HDFS

Use Load statement to load the data into a relation .

As keyword used to declare column names, as we dont have any columns, we declared only one column named line.

```
input = LOAD '/path/to/file/' AS(line:Chararray);
```

2. Convert the Sentence into words.

The data we have is in sentences. So we have to convert that data into words using TOKENIZE Function.

```
(TOKENIZE(line));
```

(or)

If we have any delimeter like space we can specify as

```
(TOKENIZE(line,' '));
```

Output will be like this:

({(This),(is),(a),(hadoop),(class)})
({(hadoop),(is),(a),(bigdata),(technology)})

but we have to convert it into multiple rows like below

(This)
(is)
(a)
(hadoop)
(class)
(hadoop)
(is)
(a)
(bigdata)
(technology)

3.Convert Column into Rows

I mean we have to convert every line of data into multiple rows ,for this we have function called
FLATTEN in pig.

Using FLATTEN function the bag is converted into tuple, means the array of strings converted into multiple rows.

Words = FOREACH input GENERATE FLATTEN(TOKENIZE(line,' ')) AS word;

Then the ouput is like below

(This)
(is)
(a)
(hadoop)
(class)
(hadoop)
(is)

(a)
(bigdata)
(technology)

3. Apply GROUP BY

We have to count each word occurance, for that we have to group all the words.

Grouped = GROUP words BY word;

4. Generate word count

wordcount = FOREACH Grouped GENERATE group, COUNT(words);

We can print the word count on console using Dump.

DUMP wordcount;

Output will be like below.

(a,2)
(is,2)
(This,1)
(class,1)
(hadoop,2)
(bigdata,1)
(technology,1)

Below is the complete program for the same.

```
input = LOAD '/path/to/file/' AS(line:Chararray);
Words = FOREACH input GENERATE FLATTEN(TOKENIZE(line,' ')) AS word;
Grouped = GROUP words BY word;
wordcount = FOREACH Grouped GENERATE group, COUNT(words);
```

TASK-7

Aim: Writing User Defined Functions/Eval functions for filtering unwanted data in Pig

Apache Pig provides extensive support for **U**ser **D**efined **F**unctions (UDF's). Using these UDF's, we can define our own functions and use them. The UDF support is provided in six programming languages, namely, Java, Jython, Python, JavaScript, Ruby and Groovy.

For writing UDF's, complete support is provided in Java and limited support is provided in all the remaining languages. Using Java, you can write UDF's involving all parts of the processing like data load/store, column transformation, and aggregation. Since Apache Pig has been written in Java, the UDF's written using Java language work efficiently compared to other languages.

In Apache Pig, we also have a Java repository for UDF's named **Piggybank**. Using Piggybank, we can access Java UDF's written by other users, and contribute our own UDF's.

Types of UDF's in Java

While writing UDF's using Java, we can create and use the following three types of functions −

- **Filter Functions** − The filter functions are used as conditions in filter statements. These functions accept a Pig value as input and return a Boolean value.
- **Eval Functions** − The Eval functions are used in FOREACH-GENERATE statements. These functions accept a Pig value as input and return a Pig result.
- **Algebraic Functions** − The Algebraic functions act on inner bags in a FOREACHGENERATE statement. These functions are used to perform full MapReduce operations on an inner bag.

Writing UDF's using Java

To write a UDF using Java, we have to integrate the jar file **Pig-0.15.0.jar**. In this section, we discuss how to write a sample UDF using Eclipse. Before proceeding further, make sure you have installed Eclipse and Maven in your system.

Follow the steps given below to write a UDF function −

- Open Eclipse and create a new project (say **myproject**).

- Convert the newly created project into a Maven project.
- Copy the following content in the pom.xml. This file contains the Maven dependencies for Apache Pig and Hadoop-core jar files.

```
<project xmlns = "http://maven.apache.org/POM/4.0.0"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0http://maven.apache
.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>Pig_Udf</groupId>
  <artifactId>Pig_Udf</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.3</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <dependencies>

    <dependency>
      <groupId>org.apache.pig</groupId>
      <artifactId>pig</artifactId>
      <version>0.15.0</version>
    </dependency>

    <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-core</artifactId>
      <version>0.20.2</version>
```

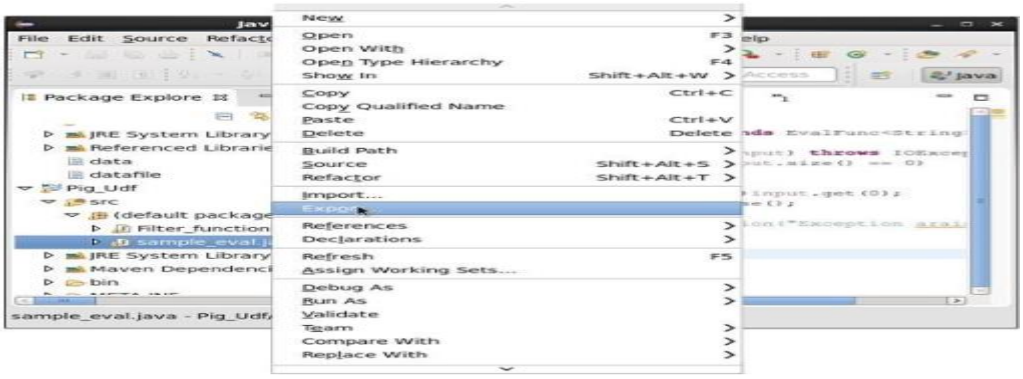
```
    </dependency>

  </dependencies>

</project>
```

- Save the file and refresh it. In the **Maven Dependencies** section, you can find the downloaded jar files.
- Create a new class file with name **Sample_Eval** and copy the following content in it.

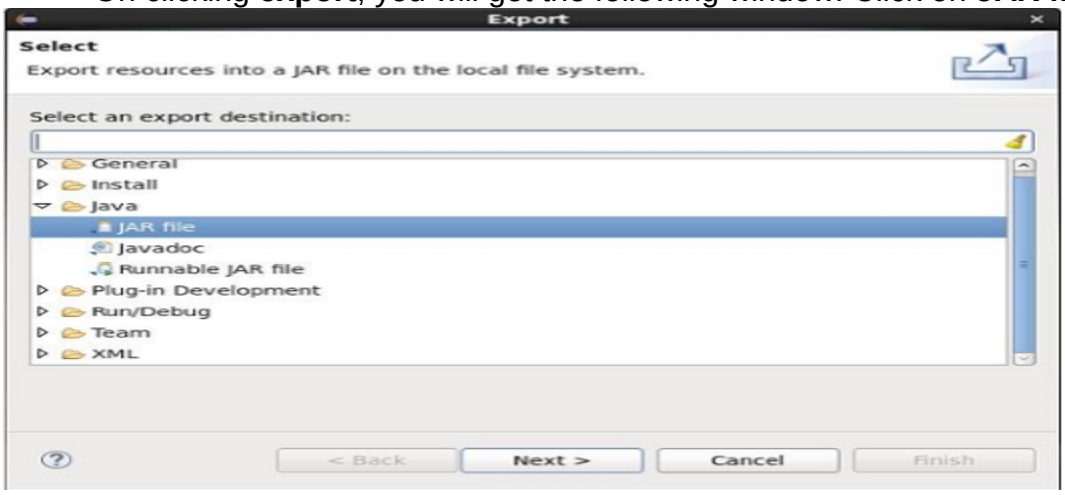
```
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;

import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;

public class Sample_Eval extends EvalFunc<String>{

  public String exec(Tuple input) throws IOException {
    if (input == null || input.size() == 0)
    return null;
    String str = (String)input.get(0);
    return str.toUpperCase();
  }
}
```

While writing UDF's, it is mandatory to inherit the EvalFunc class and provide implementation to **exec()** function. Within this function, the code required for the UDF is written. In the above example, we have return the code to convert the contents of the given column to uppercase.

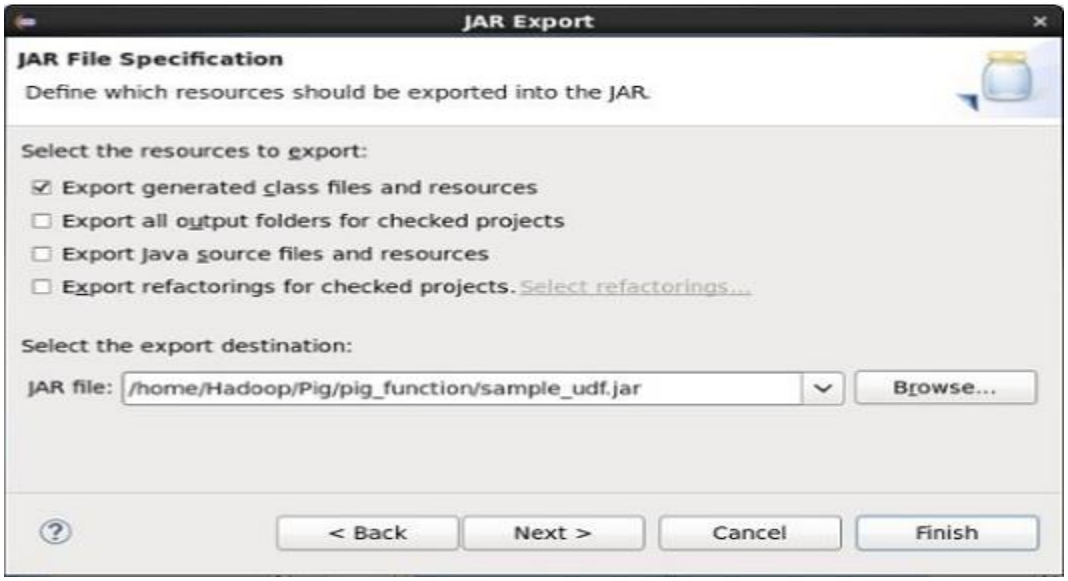
- After compiling the class without errors, right-click on the Sample_Eval.java file. It gives you a menu. Select **export** as shown in the following screenshot.



- On clicking **export**, you will get the following window. Click on **JAR file**.



- Proceed further by clicking **Next>** button. You will get another window where you need to enter the path in the local file system, where you need to store the jar file.



- Finally click the **Finish** button. In the specified folder, a Jar file **sample_udf.jar** is created. This jar file contains the UDF written in Java.

Using the UDF

After writing the UDF and generating the Jar file, follow the steps given below −

Step 1: Registering the Jar file

After writing UDF (in Java) we have to register the Jar file that contain the UDF using the Register operator. By registering the Jar file, users can intimate the location of the UDF to Apache Pig.

Syntax

Given below is the syntax of the Register operator.

REGISTER path;

Example

As an example let us register the sample_udf.jar created earlier in this chapter.

Start Apache Pig in local mode and register the jar file sample_udf.jar as shown below.

$cd PIG_HOME/bin

$./pig –x local

REGISTER '/$PIG_HOME/sample_udf.jar'

Note − assume the Jar file in the path − /$PIG_HOME/sample_udf.jar

Step 2: Defining Alias

After registering the UDF we can define an alias to it using the **Define** operator.

Syntax

Given below is the syntax of the Define operator.

DEFINE alias {function | [`command` [input] [output] [ship] [cache] [stderr]] };

Example

Define the alias for sample_eval as shown below.

DEFINE sample_eval sample_eval();

Step 3: Using the UDF

After defining the alias you can use the UDF same as the built-in functions. Suppose there is a file named emp_data in the HDFS **/Pig_Data/** directory with the following content.

001,Robin,22,newyork
002,BOB,23,Kolkata
003,Maya,23,Tokyo
004,Sara,25,London
005,David,23,Bhuwaneshwar
006,Maggy,22,Chennai
007,Robert,22,newyork
008,Syam,23,Kolkata
009,Mary,25,Tokyo
010,Saran,25,London
011,Stacy,25,Bhuwaneshwar
012,Kelly,22,Chennai

And assume we have loaded this file into Pig as shown below.

```
grunt> emp_data = LOAD 'hdfs://localhost:9000/pig_data/emp1.txt' USING PigStorage(',')
   as (id:int, name:chararray, age:int, city:chararray);
```

Let us now convert the names of the employees in to upper case using the UDF **sample_eval**.

```
grunt> Upper_case = FOREACH emp_data GENERATE sample_eval(name);
```

Verify the contents of the relation **Upper_case** as shown below.

grunt> Dump Upper_case;

(ROBIN)
(BOB)
(MAYA)
(SARA)
(DAVID)
(MAGGY)
(ROBERT)
(SYAM)
(MARY)
(SARAN)
(STACY)
(KELLY)

TASK-8

Aim: Working with Hive QL, Use Hive to create, alter, and drop databases, tables, view functions, and indexes

1. CREATE
2. SHOW
3. DESCRIBE
4. USE
5. DROP
6. ALTER
7. TRUNCATE

Table-1 Hive DDL commands

DDL Command	Use With
CREATE	Database, Table
SHOW	Databases, Tables, Table Properties, Partitions, Functions, Index
DESCRIBE	Database, Table, view
USE	Database
DROP	Database, Table
ALTER	Database, Table
TRUNCATE	Table

Before moving forward, note that the Hive commands are **case-insensitive**.
CREATE DATABASE is the same as create database.

So now, let us go through each of the commands deeply. Let's start with the DDL commands on Databases in Hive.

DDL Commands On Databases in Hive

1. CREATE DATABASE in Hive

The **CREATE DATABASE** statement is used to create a database in the Hive. The DATABASE and SCHEMA are interchangeable. We can use either DATABASE or SCHEMA.

Syntax:

CREATE (DATABASE|SCHEMA) [**IF** NOT EXISTS] database_name

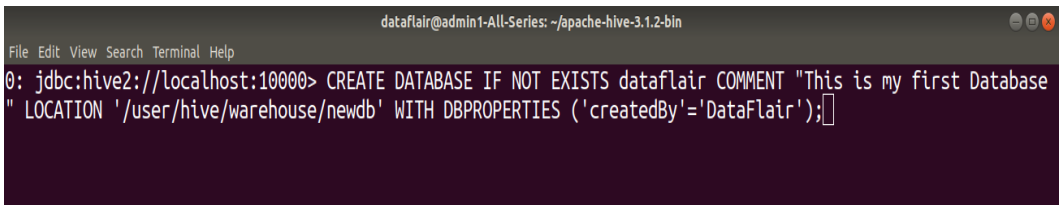
[COMMENT database_comment]

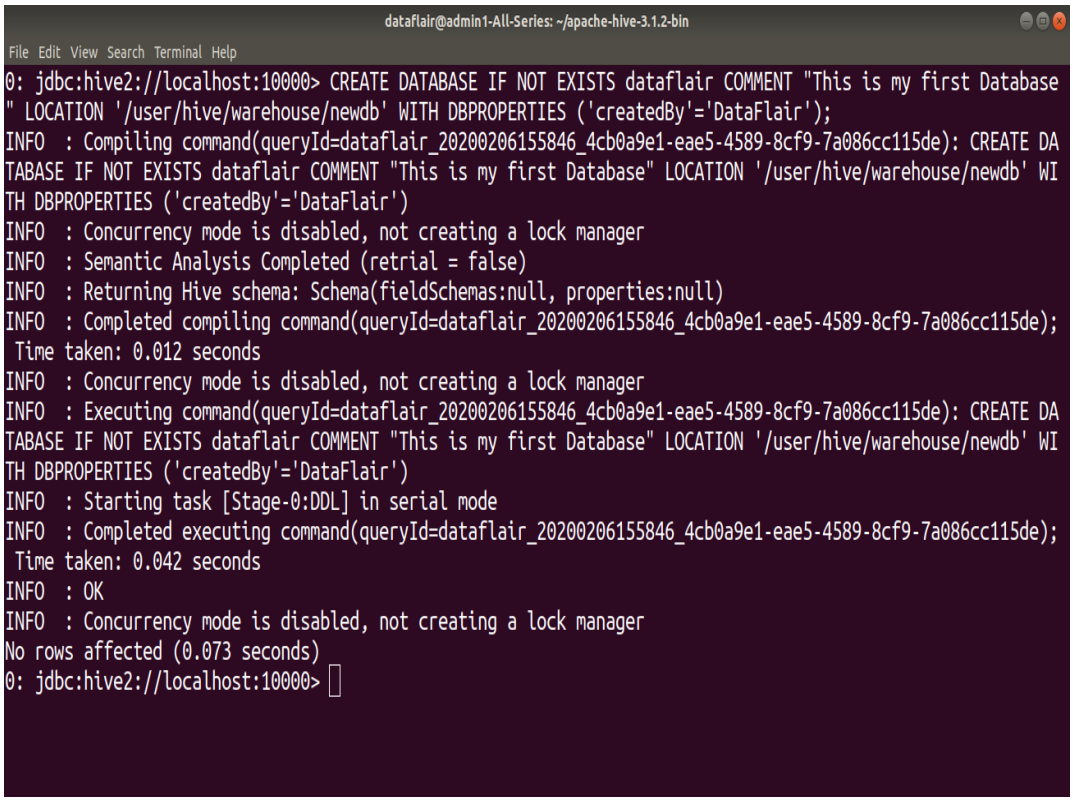
[LOCATION hdfs_path]

[WITH DBPROPERTIES (property_name=property_value, ...)];

DDL CREATE DATABASE Example:

Here in this example, we are creating a database 'dataflair'.



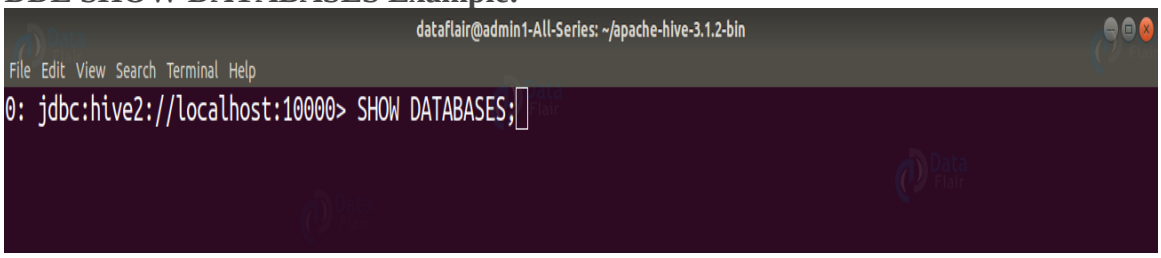


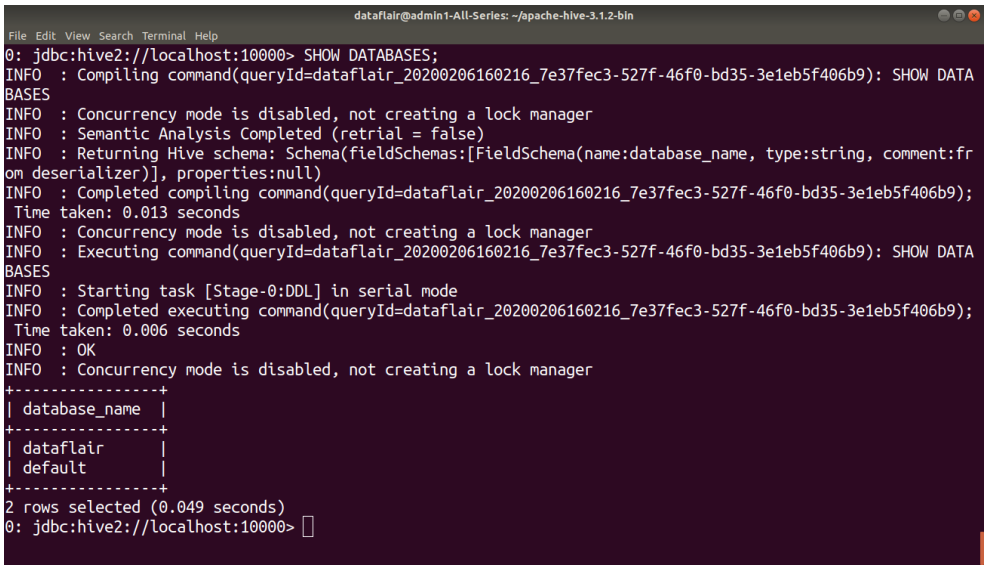
2. SHOW DATABASE in Hive

The **SHOW DATABASES** statement lists all the databases present in the Hive.
Syntax:

SHOW (DATABASES|SCHEMAS);

DDL SHOW DATABASES Example:



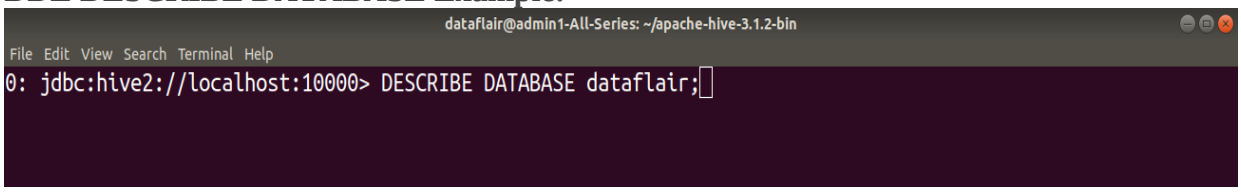


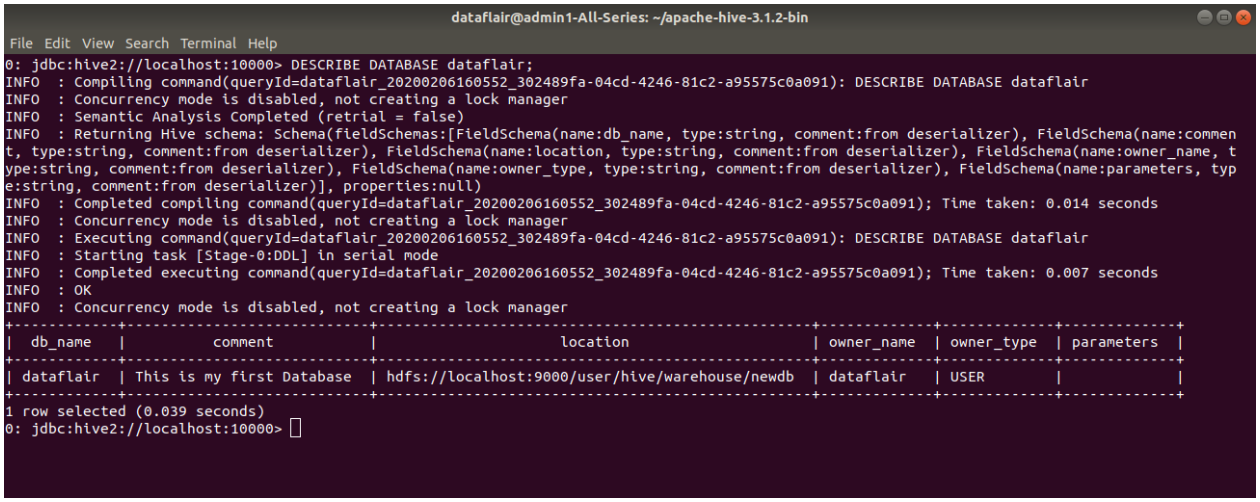
3. DESCRIBE DATABASE in Hive

The **DESCRIBE DATABASE** statement in Hive shows the name of Database in Hive, its comment (if set), and its location on the file system.
The **EXTENDED** can be used to get the database properties.
Syntax:

DESCRIBE DATABASE/SCHEMA [EXTENDED] db_name;

DDL DESCRIBE DATABASE Example:





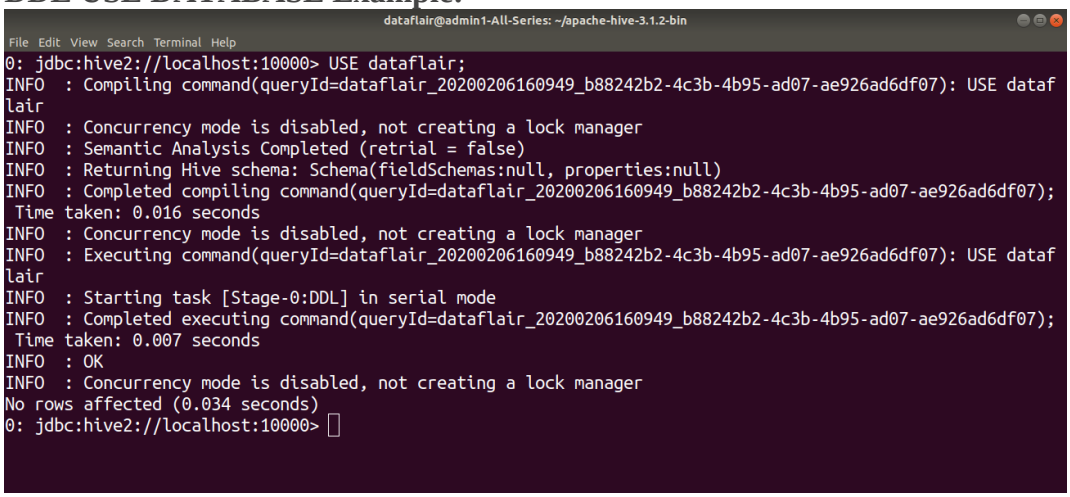
4. USE DATABASE in Hive

The **USE** statement in Hive is used to select the specific database for a session on which all subsequent HiveQL statements would be executed.

Syntax:

USE database_name;

DDL USE DATABASE Example:



5. DROP DATABASE in Hive

The **DROP DATABASE** statement in Hive is used to Drop (delete) the database.

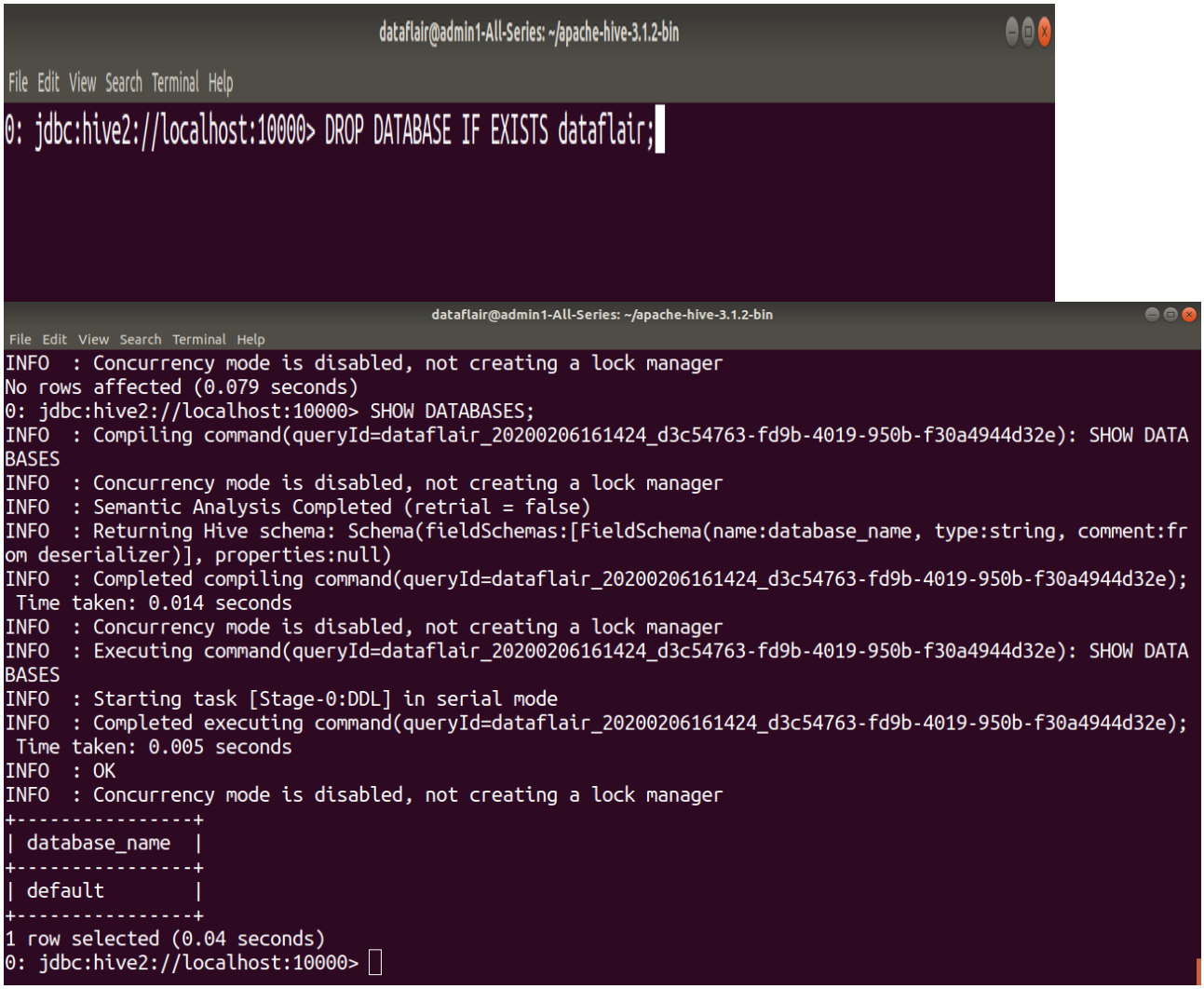
The default behavior is RESTRICT which means that the database is dropped only when it is empty. To drop the database with tables, we can use CASCADE.

Syntax:

DROP (DATABASE|SCHEMA) [IF EXISTS] database_name [RESTRICT|CASCADE];

DDL DROP DATABASE Example:

Here in this example, we are dropping a database 'dataflair' using the DROP statement.



6. ALTER DATABASE in Hive

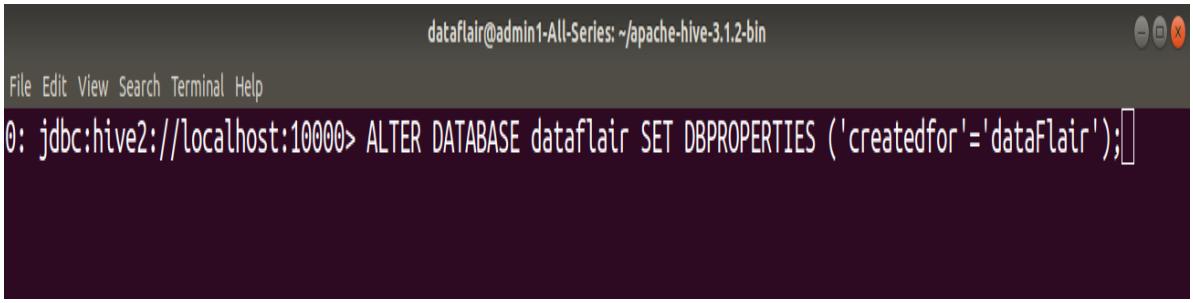
The **ALTER DATABASE** statement in Hive is used to change the metadata associated with the database in Hive.

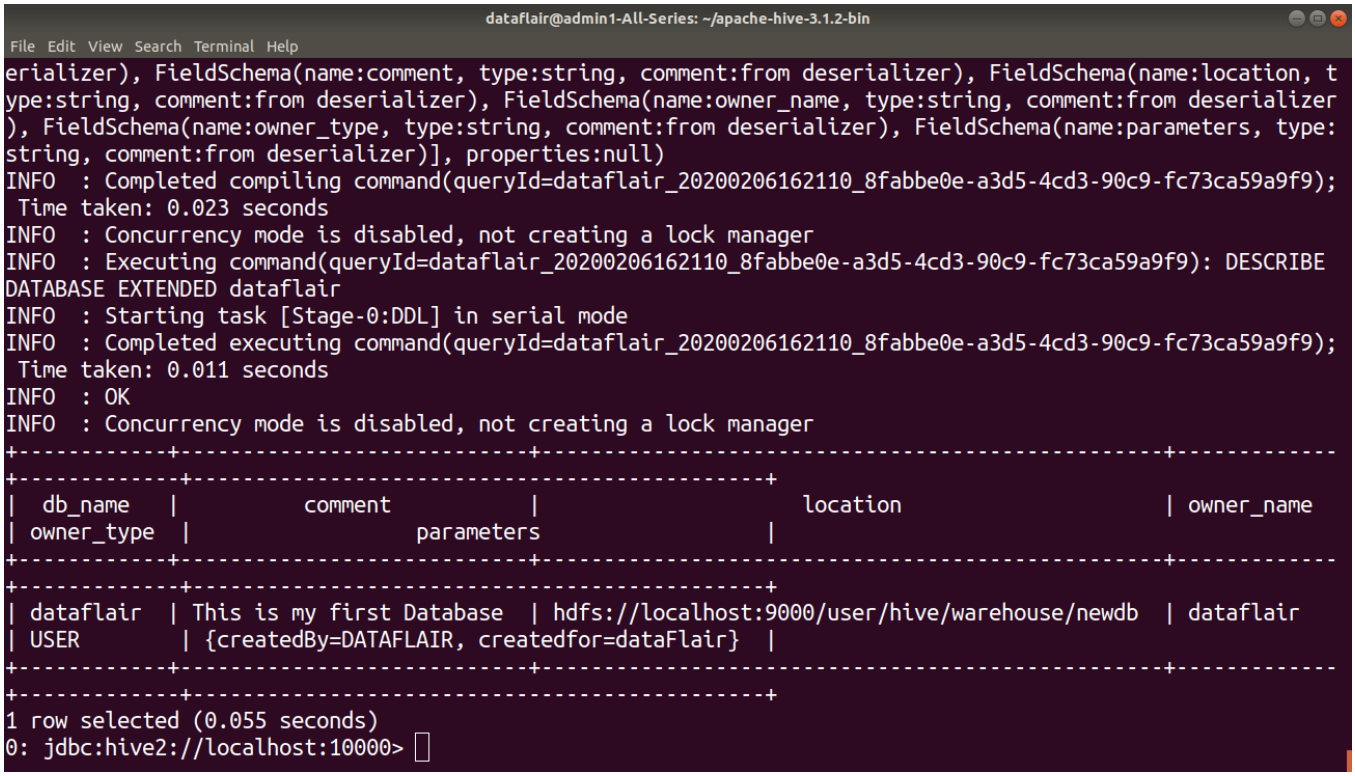
Syntax for changing Database Properties:

ALTER (DATABASE|SCHEMA) database_name SET DBPROPERTIES (property_name=property_value, ...);

DDL ALTER DATABASE properties Example:

In this example, we are setting the database properties of the 'dataflair' database after its creation by using the ALTER command.



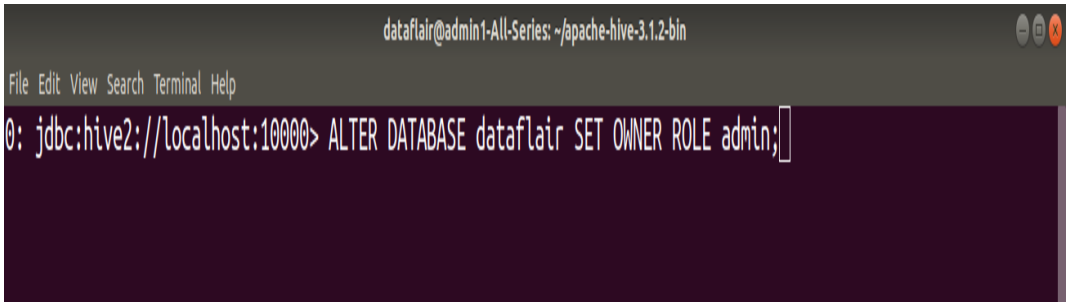


Syntax for changing Database owner:

ALTER (DATABASE|SCHEMA) database_name SET OWNER [USER|ROLE] user_or_role;

DDL ALTER DATABASE owner Example:

In this example, we are changing the owner role of the 'dataflair' database using the ALTER statement.



Syntax for changing Database Location:
ALTER (DATABASE|SCHEMA) database_name SET LOCATION hdfs_path;

Note: The ALTER DATABASE … SET LOCATION statement does not move the database current directory contents to the newly specified location.
This statement does not change the locations associated with any tables or partitions under the specified database. Instead, it changes the default parent-directory, where new tables will be added for this database.

No other metadata associated with the database can be changed.

DDL Commands on Tables in Hive

1. CREATE TABLE

The **CREATE TABLE** statement in Hive is used to create a table with the given name. If a table or view already exists with the same name, then the error is thrown. We can use **IF NOT EXISTS** to skip the error.
Syntax:
CREATE TABLE [**IF** NOT EXISTS] [db_name.] table_name [(col_name data_type [COMMENT col_comment], ... [COMMENT col_comment])] [COMMENT table_comment] [ROW FORMAT row_format] [STORED AS file_format] [LOCATION hdfs_path];

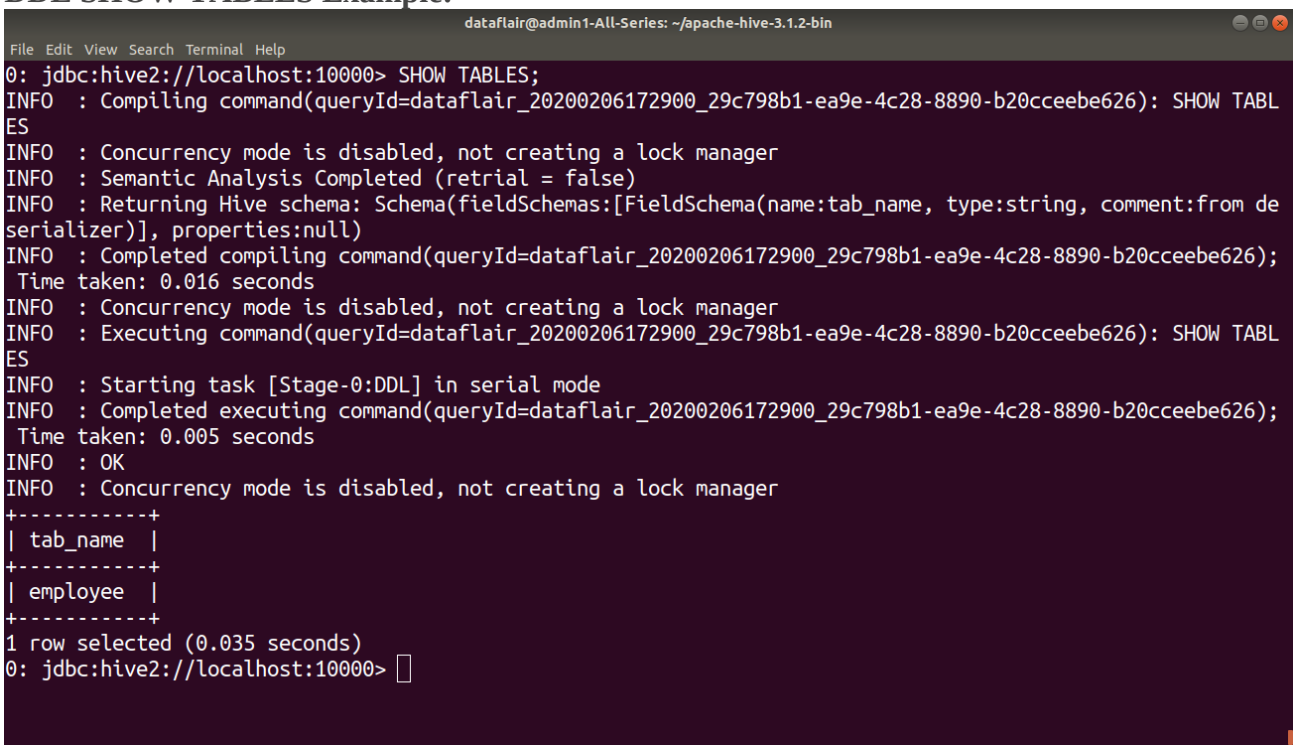
DDL CREATE TABLE Example:
In this table, we are creating a table 'Employee' in the 'dataflair' database.

ROW FORMAT DELIMITED means we are telling the Hive that when it finds a new line character, that means a new record.
FIELDS TERMINATED BY ',' tells Hive what delimiter we are using in our files to separate each column.
STORED AS TEXTFILE is to tell Hive what type of file to expect.
2. SHOW TABLES in Hive

The **SHOW TABLES** statement in Hive lists all the base tables and **views** in the current database.
Syntax:

SHOW TABLES [IN database_name];

DDL SHOW TABLES Example:



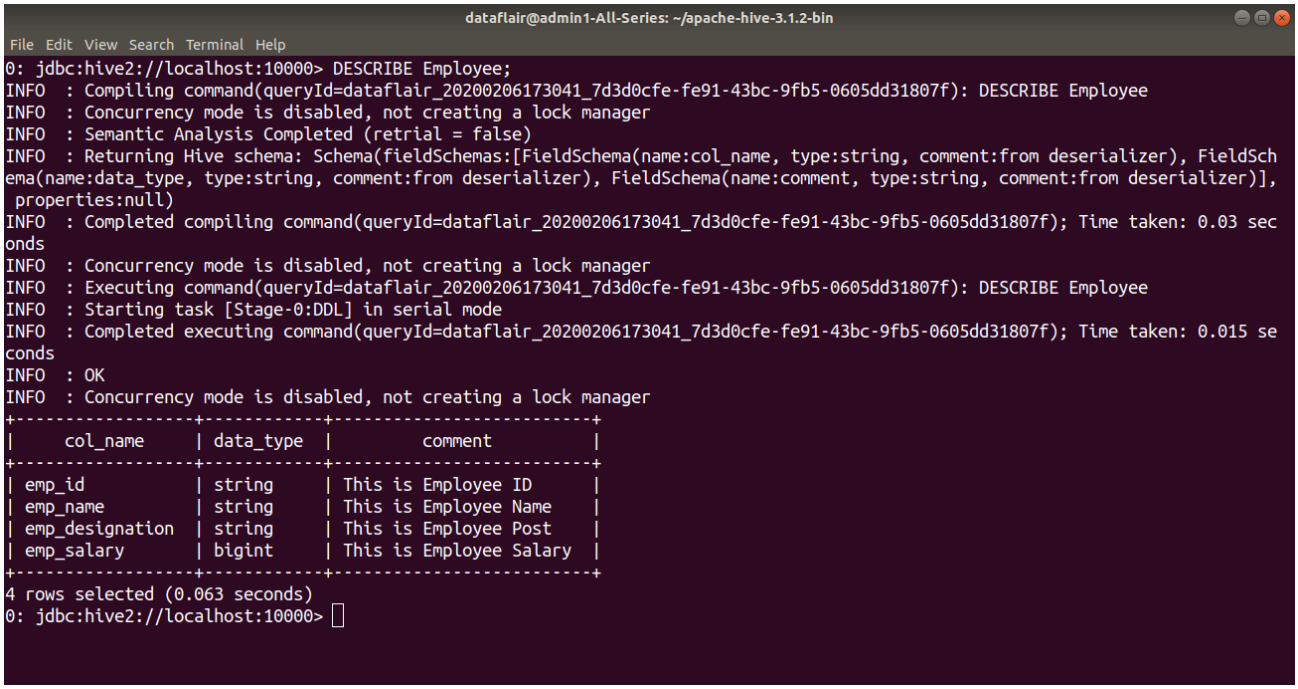
3. DESCRIBE TABLE in Hive

The **DESCRIBE** statement in Hive shows the lists of columns for the specified table.

Syntax:

DESCRIBE [EXTENDED|FORMATTED] [db_name.] table_name[.col_name ([.field_name])];

DDL DESCRIBE TABLE Example:



4. DROP TABLE in Hive

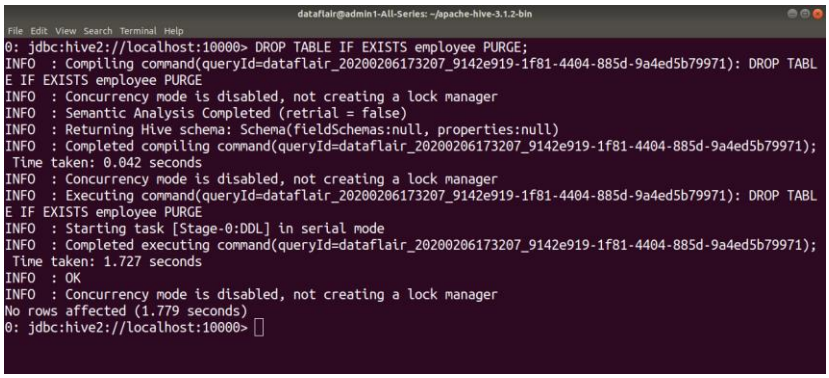
The **DROP TABLE** statement in Hive deletes the data for a particular table and remove all metadata associated with it from Hive metastore.
If **PURGE** is not specified then the data is actually moved to the .Trash/current directory.
If **PURGE** is specified, then data is lost completely.

Syntax:

DROP TABLE [**IF** EXISTS] table_name [PURGE];

DDL DROP TABLE Example:

In the below example, we are deleting the 'employee' table.



5. ALTER TABLE in Hive

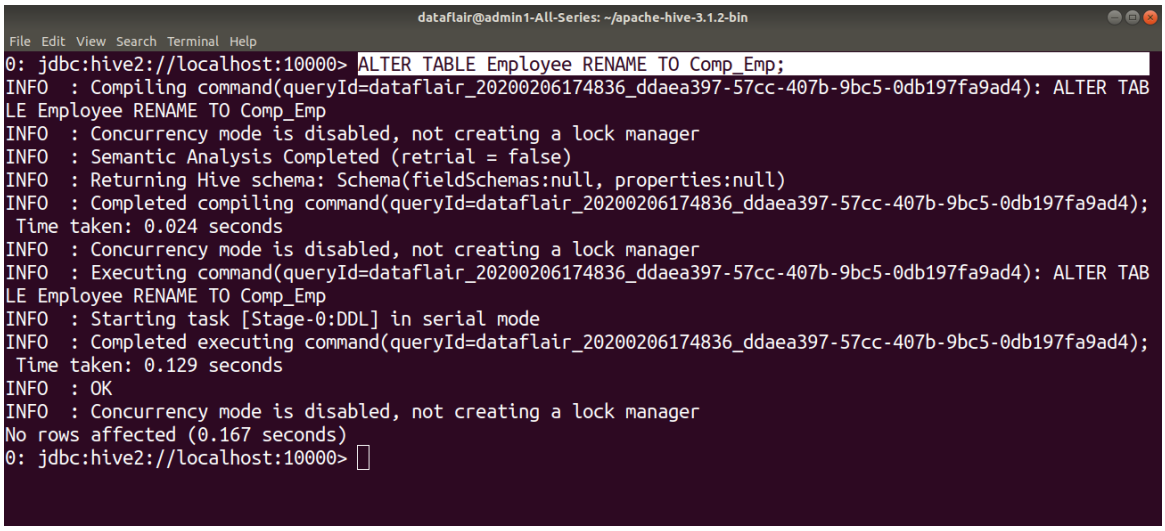
The **ALTER TABLE** statement in Hive enables you to change the structure of an existing table. Using the ALTER TABLE statement we can rename the table, add columns to the table, change the table properties, etc.

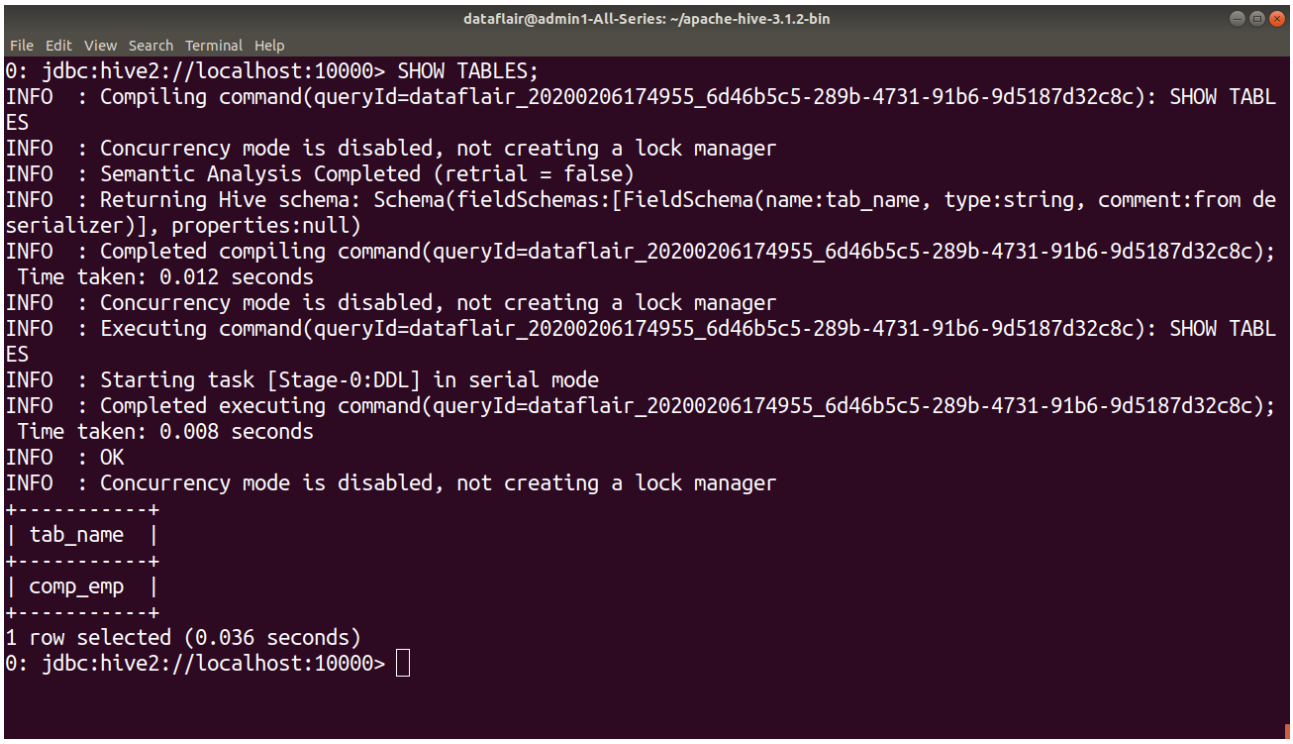
Syntax to Rename a table:

ALTER TABLE table_name RENAME TO new_table_name;

DDL ALTER TABLE name Example:

In this example, we are trying to rename the 'Employee' table to 'Com_Emp' using the ALTER statement.



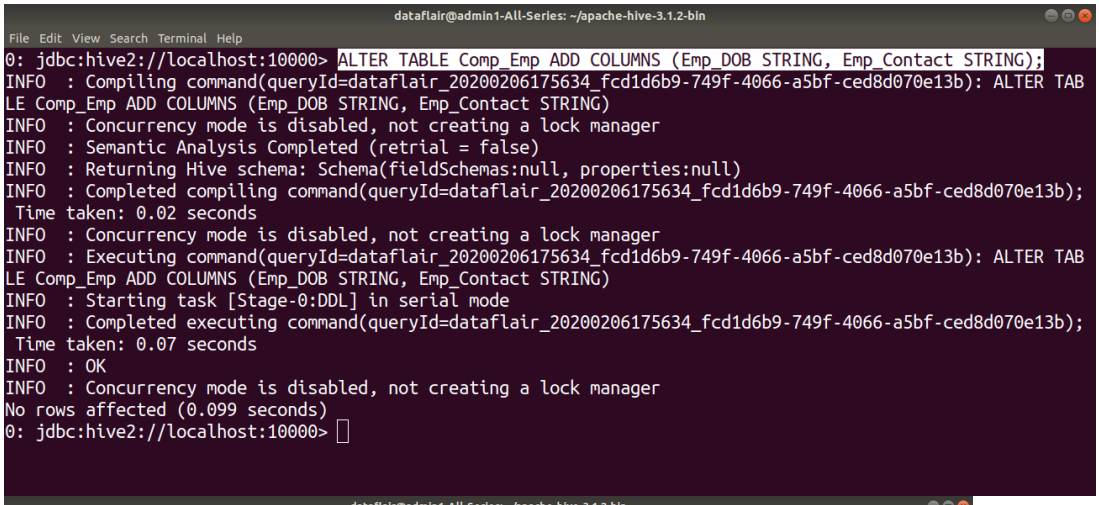


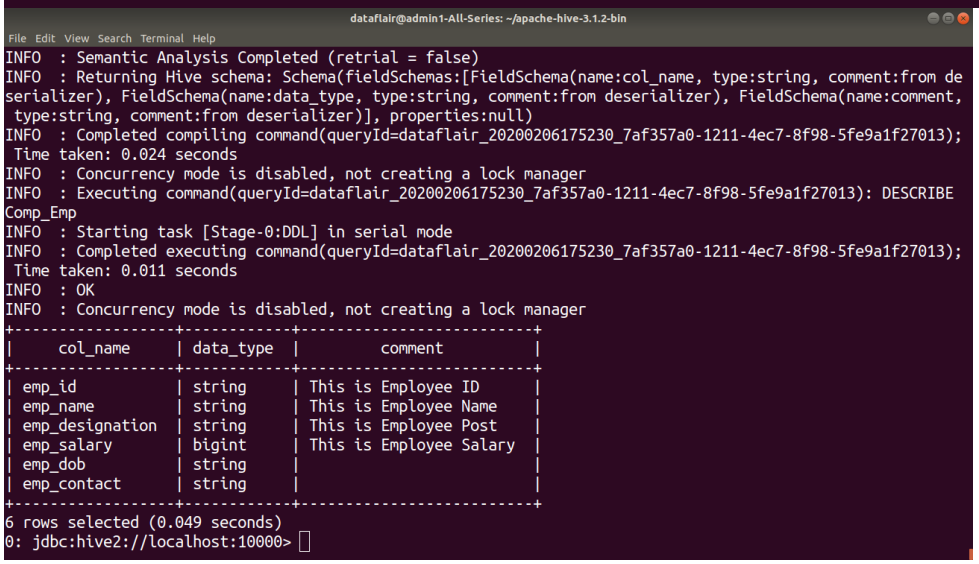
Syntax to Add columns to a table:

ALTER TABLE table_name ADD COLUMNS (column1, column2) ;

DDL ALTER TABLE columns Example:

In this example, we are adding two columns 'Emp_DOB' and 'Emp_Contact' in the 'Comp_Emp' table using the ALTER command.





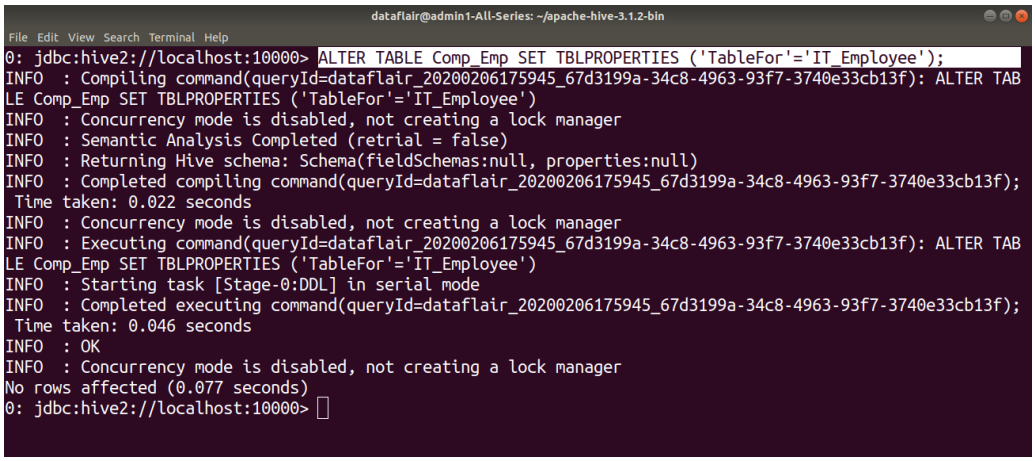
Syntax to set table properties:

ALTER TABLE table_name SET TBLPROPERTIES

(‘property_key’=’property_new_value’);

DDL ALTER TABLE properties Example:

In this example, we are setting the table properties after table creation by using ALTER command.

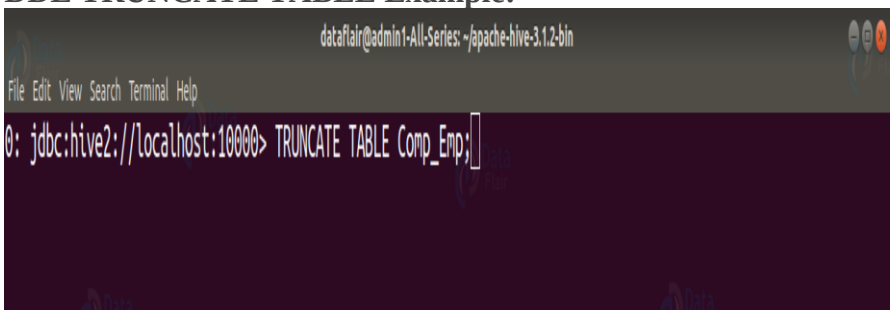


6. TRUNCATE TABLE

TRUNCATE TABLE statement in Hive removes all the rows from the table or partition.
Syntax:

TRUNCATE TABLE table_name;

DDL TRUNCATE TABLE Example:



TASK 9

Aim: Writing User Defined Functions in Hive
Procedure:

UDF (User Defined Function)

Here I am going to show how to write a simple "trim-like" function called "Strip" – of course, you can write something fancier, but my goal here is to take away something in a short amount of time. So let's begin.

How to Write a UDF function in Hive?

1. Create a Java class for the User Defined Function which extends ora.apache.hadoop.hive.sq.exec.UDF and implements more than one evaluate() methods. Put in your desired logic and you are almost there.
2. Package your Java class into a JAR file (I am using Maven)
3. Go to Hive CLI, add your JAR, and verify your JARs is in the Hive CLI classpath
4. CREATE TEMPORARY FUNCTION in Hive which points to your Java class
5. Use it in Hive SQL and have fun!

There are better ways to do this, by writing your own GenericUDF to deal with non-primitive types like arrays and maps – but I am not going to cover it in this article.

I will go into detail for each one.

Create Java Class for a User Defined Function

As you can see below I am calling my Java class "Strip". You can call it anything, but the important point is that it extends the UDF interface and provides two evaluate() implementations.

evaluate(Text str, String stripChars) - will trim specified characters in stripChars from first argument str.
evaluate(Text str) - will trim leading and trailing spaces

```
package org.hardik.letsdobigdata;
import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;
```

```
public class Strip extends UDF {

private Text result = new Text();
 public Text evaluate(Text str, String stripChars) {
 if(str == null) {
 return null;
 }
 result.set(StringUtils.strip(str.toString(), stripChars));
 return result;
 }
 public Text evaluate(Text str) {
 if(str == null) {
 return null;
 }
 result.set(StringUtils.strip(str.toString()));
 return result;
 }
}
```

Package Your Java Class into a JAR

There is a pom.xml attached in GitHub. Please make sure you have Maven installed. If you are working with a GitHub clone, go to your shell:

$ cd HiveUDFs
and run "mvn clean package". This will create a JAR file which contains our UDF class. Copy the JAR's path.

Go to the Hive CLI and Add the UDF JAR

hive> ADD /home/cloudera/workspace/HiveUDFs/target/HiveUDFs-0.0.1-SNAPSHOT.jar;
Added [/home/cloudera/workspace/HiveUDFs/target/HiveUDFs-0.0.1-SNAPSHOT.jar] to class path

Added resources: [/home/cloudera/workspace/HiveUDFs/target/HiveUDFs-0.0.1-SNAPSHOT.jar]

Verify JAR is in Hive CLI Classpath

You should see your jar in the list.

```
hive> list jars;
/usr/lib/hive/lib/hive-contrib.jar
/home/cloudera/workspace/HiveUDFs/target/HiveUDFs-0.0.1-SNAPSHOT.jar
```

Create Temporary Function

It does not have to be a temporary function. You can create your own function, but just to keep things moving, go ahead and create a temporary function.

You may want to add ADD JAR and CREATE TEMPORARY FUNCTION to .hiverc file so they will execute at the beginning of each Hive session.

UDF Output

The first query strips ‘ha’ from string ‘hadoop’ as expected (2 argument evaluate() in code). The second query strips trailing and leading spaces as expected.

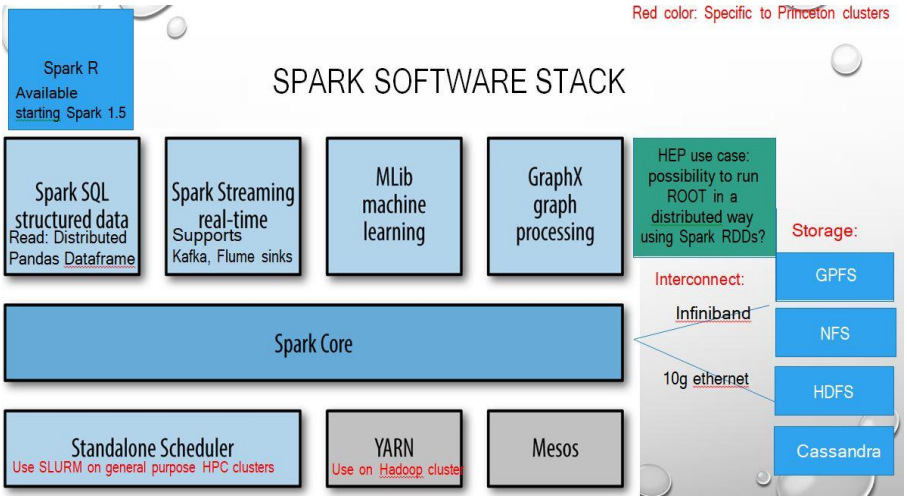
```
hive> CREATE TEMPORARY FUNCTION STRIP AS 'org.hardik.letsdobigdata.Strip';
 hive> select strip('hadoop','ha') from dummy;
 OK
 doop
 Time taken: 0.131 seconds, Fetched: 1 row(s)
 hive> select strip('   hiveUDF ') from dummy;
 OK
 hiveUDF
```

TASK 10

Aim: Understanding the processing of large dataset on Spark framework.

Large-Scale Data Analysis With Apache Spark

• Apache Spark Is A Fast And General Purpose Cluster Computing Framework For Large-Scale Data Processing
• It Became A De-Facto Industry Standard For Data Analysis, Replacing Mapreduce Computing Engine
• Mapreduce Engine Is Going To Be Retired By Cloudera - A Major Hadoop Distribution Provider – Starting the Version Cdh5.5
• Spark Does Not Use The Mapreduce As An Execution Engine, However, It Is Closely Integrated With Hadoop Ecosystem And Can Be Run Via Yarn, Use The Hadoop File Formats, And Hdfs Storage
• On The Other Hand, It Can Be Used In A Standalone Mode On Any Hpc Clusters
• E.G. Via Slurm Resource Manager As It Is Done At Princeton
• Spark Is Best Known For Its Ability To Persist Large Datasets In Memory Between Jobs
• Spark Is Written In Scala, But There Are Language Bindings For Python, Scala, And Java



Programming With Rdds (I)

•Rdds (Resilient Distributed Datasets) Are Read-Only Partitioned Collections Of Objects

•Each Rdd Is Split Into Partitions Which Can Be Computed On Different Nodes Of A Cluster

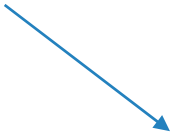
•Partitions Define The Level Of Parallelism In A Spark App: Important Parameter To Tune!

•Create An Rdd By Loading Data Into It, Or Parallelizing Existing Collection Of Objects (List, Set…)

```
npartitions = 10
sc = SparkContext(master, "TestApp")
lines = sc.parallelize(["pandas", "i like pandas"],npartitions)
```

When data has no parent RDD/input file the # of partitionsis set according to the total number of cores on nodes which run executors on them.

```
npartitions = 10
sc = SparkContext(master, "TestApp")
lines = sc.textFile("/user/alexeys/test.txt",npartitions)
```



Spark automatically sets the # of partitions according to the number of file system block the file spans over. For reducetasks, it is set according to the parent RDD

Transformations: Operations On Rdd That Return A New Rdd. They Do Not Mutate The Old Rdd, But Rather Return A Pointer To It.

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.

Actions: Actions Force Program To Produce Some Outpout (Note: Rdds Are Lazily Evaluated)

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.

• Lazy Evaluation: Rdd Transformations Are Not Evaluated Until an Action Is Called On It
• Persistance/Caching: Unlike Mapreduce, Where Making An Intermediate Result Obtained On The Entire Dataset Available To All Nodes Would Only Be Possible By Splitting The Calculation Into Multiple Map-Reduce Stages (Chaining) And Performing An Intermediate Shuffle
•Crucial Feature For Iterative Algorithms Like, For Instance, K-Means

•Spark Allows To Persist Datasets In Memory As Well As Memory/Disk (Split In A Specified Proportion Controlled In Config)
•Pyspark Uses Cpickle For Serializing Data

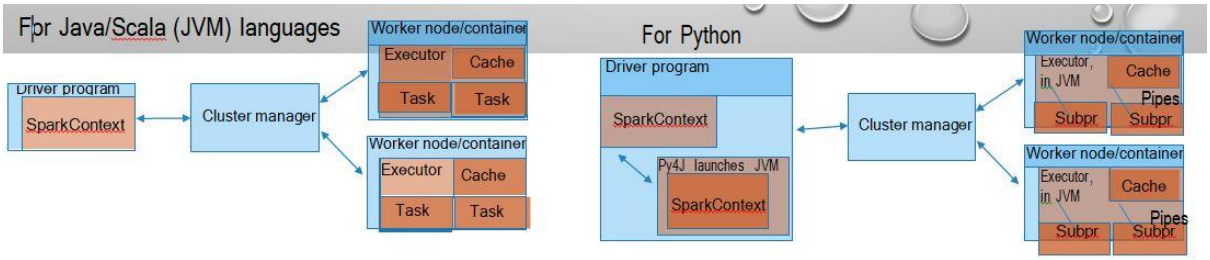
ANATOMY OF A SPARK APP: RUNING ON A CLUSTER (I)

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions t don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient th deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Store RDD in serialized format in Tachyon. Compared to MEMORY_ONLY_SER, OFF_HEAP reduces garba collection overhead and allows executors to be smaller and to share a pool of memory, making it attractive environments with large heaps or multiple concurrent applications. Furthermore, as the RDDs reside in Tachyon, the crash of an executor does not lead to losing the in-memory cache. In this mode, the memory Tachyon is discardable. Thus, Tachyon does not attempt to reconstruct a block that it evicts from memory. you plan to use Tachyon as the off heap store, Spark is compatible with Tachyon out-of-the-box. Please re to this page for the suggested version pairings.

From Spark Manua

ANATOMY OF A SPARK APP: RUNING ON A CLUSTER (I)

•Spark Uses Master/Slave Architecture With One Central Coordinator (Driver) And Many Distribtued Workers (Executors)
•Driver Runs Its Own Java Process, Executors Each Run Their Own Java Processes
•For Pyspark, Sparkcontext Uses Py4j To Launch A Jvm And Create A Javasparkcontext
•Executors All Ran In Jvms, And Python Subprocesses (Tasks) Which Are Launched Communicate With Them Using Pipes



SPARK USER EXPERIENCE AT PRINCETON

•Most Of The Current Spark Users Come From Cs And Politics Departments
•We Started Out By Using Spark Via Yarn Installed As A Part Of The Cloudera Hadoop Distribution (Still Available On The Bigdata Cluster)
•Switched To Spark In A Standalone Mode Via Slurm On General Hpc Clusters
•Yarn Uses Containers (Slurm Will Soon Too…), Allows Dynamic Allocation
•Slurm Is A Better Choice For Us Because Our Clusters Are Not Pure Spark Or Hadoop Clusters, And Resources Need To Be Shared
•Slurm Solution Is Rather Mature: Allows Allocation Of Multiple Executors Per Node
•Most Difficulties For A User Is In Memory Allocation
•See The Following Resources For More Details On Spark+Slurm

ANALYSIS EXAMPLE WITH SPARK, SPARK SQL AND MLLIB

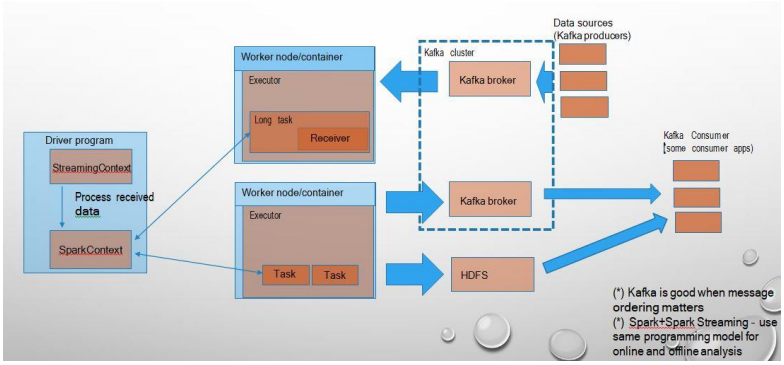
```
def main(argv):
    #STEP1: data ingestion
    sc = SparkContext(appName="KaggleDato_Step2")
    sqlContext = SQLContext(sc)

    #read data into RDD
    input_schema_rdd = sqlContext.read.json("file:///scratch/network/alexeys/KaggleDato/Preprocessed/0_1/part-00000")

    train_label_rdd = sqlContext.read.json(PATH_TO_TRAIN_LABELS)
    sub_label_rdd = sqlContext.read.json(PATH_TO_SUB_LABELS)

    input_schema_rdd.registerTempTable("input")
    train_label_rdd.registerTempTable("train_label")
    sub_label_rdd.registerTempTable("sub_label")

    #Split into 2 subsamples with different label for classification
    train_wlabels_0 = sqlContext.sql("SELECT title,text,images,links,label FROM input JOIN train_label WHERE input.id = train_label.id AND label = 0")
    train_wlabels_1 = sqlContext.sql("SELECT title,text,images,links,label FROM input JOIN train_label WHERE input.id = train_label.id AND label = 1")
    sub_wlabels = sqlContext.sql("SELECT title,text,images,links,label FROM input JOIN sub_label WHERE input.id = sub_label.id")

    text_only_0 = train_wlabels_0.map(lambda p: p.text)
    text_only_1 = train_wlabels_1.map(lambda p: p.text)
    image_only_0 = train_wlabels_0.map(lambda p: p.images)
    image_only_1 = train_wlabels_1.map(lambda p: p.images)
    links_only_0 = train_wlabels_0.map(lambda p: p.links)
    links_only_1 = train_wlabels_1.map(lambda p: p.links)
    title_only_0 = train_wlabels_0.map(lambda p: p.title)
    title_only_1 = train_wlabels_1.map(lambda p: p.title)

    tf = HashingTF(numFeatures=10)
    #preprocess text features
    text_documents_0 = text_only_0.map(lambda line: tokenize(line)).map(lambda word: tf.transform(word))
    text_documents_1 = text_only_1.map(lambda line: tokenize(line)).map(lambda word: tf.transform(word))

    #add the adhoc non-text features
    documents_0 = text_documents_0.zip(image_only_0).zip(links_only_0).zip(title_only_0)
    documents_1 = text_documents_1.zip(image_only_1).zip(links_only_1).zip(title_only_1)

    #turn into a format expected by MLlib classifiers
    labeled_tfidf_0 = documents_0.map(lambda row: parsePoint(0,row))
    labeled_tfidf_1 = documents_1.map(lambda row: parsePoint(1,row))

    labeled_tfidf = labeled_tfidf_0.union(labeled_tfidf_1)
    labeled_tfidf.cache()

    #CV split
    (trainData, cvData) = labeled_tfidf.randomSplit([0.7, 0.3])
    trainData.cache()
    cvData.cache()

    #Try various classifiers
    model = RandomForest.trainClassifier(trainData, numClasses=2, categoricalFeaturesInfo={},
                                         numTrees=3, featureSubsetStrategy="auto",
                                         impurity='gini', maxDepth=4, maxBins=32)

    # Evaluate model on test instances and compute test error
    predictions = model.predict(cvData.map(lambda x: x.features))
    labelsAndPredictions = cvData.map(lambda lp: lp.label).zip(predictions)
    testErr = labelsAndPredictions.filter(lambda (v, p): v != p).count() / float(cvData.count())
    print('Test Error = ' + str(testErr))
    print('Learned classification forest model:')
    print(model.toDebugString())
```

•Mllib Is The Main Spark's Machine Learning Library
•It Contains Most Of The Ml Classifiers:
Load Scraped Data
•Logistic Regression, Tree/Forest Classifiers, Svm, Clustering Algorithms…
•Introduces New Data Format: Labeledpoint (For Supervised Learning)
Select/Join As In Pandas
Dataframe Are Avail Starting Spark 1.5
Feature Engineering
Train/Predict

•As An Example, Lets Us Take A Kaggle Competition:
•Https://Www.Kaggle.Com/C/Dato-Native
•Dataset For That Competition Consisted Of Over 300k Raw Html Files Containing Text, Links, And Downloadable Images
•The Challenge Was To Identify The Paid Content Disguised As Just Another Internet Gem (I.E. "Non-Native" Advertisement)
Analysis Flow:
•Scrape The Data From The Web-Pages: Text, Images, Links…
•Feature Engineering: Extract Features For Classification
•Train/Cross Validate A Machine Learning Model
•Predict

REAL-TIME ANALYSES WITH APACHE SPARK
•Many Applications Benefit From Acting On Data As Soon As It Arrives
•Not A Typical Case For Physics Analyses At Cms…
•However, It Could Be A Perfect Fit For Exotica Hotline Or Any Other "Hotline" Type Systems Or Anomaly Detection During Data Collection
•Spark Streaming Uses A Concept Of Dstreams (Seq. Of Data Arriving Over Time)
•Ingest And Analyze Data Collected Over A Batch Interval
•Supports Various Input Sources: Akka, Flume, Kafka, Hdfs
•Can Operate 24/7, But Is Not Truly Real-Time (Like E.G. Apache Storm) – It Is A Micro- Batch System With A Fixed (Controlled) Batch Interval
•Fully Fault Tolerant, Offering "Exactly Once" Semantics, So That The Data Will Be Analysed For Sure Even If A Node Fails
•Supports Checkpointing – Allowing To Restore Data From A Given Point In Time

EXAMPLE OF A REAL-TIME PIPELINE USING APACHE KAFKA AND SPARK STREAMING



TASK 11

Aim: Ingesting Structured and Unstructured Data Using Sqoop, Flume

Apache Hadoop is an open source framework for big data processing and it is made of cluster of cooperative computers powered by Map Reduce Programming model offering distributed massive parallel processing of big data. Data in Hadoop is stored using Hadoop Distributed File System (HDFS).In HDFS, data is divided into blocks and these blocks are scattered across data nodes with default replication factor of three. These data blocks are processed simultaneously by Map Reduce Job using principle of data locality. Data into HDFS is generally copied or moved using HDFS commands. However data from sources like social media, RDBMS, web server logs etc. cannot be inserted into HDFS directly and hence data ingestion tools are needed. Data Ingestion refers to loading data into HDFS from cross platform data sources like RDBMS, Social Networks etc. The objective of this project is to load data from Relation data base, social networks and process them using suitable components of the Hadoop framework.

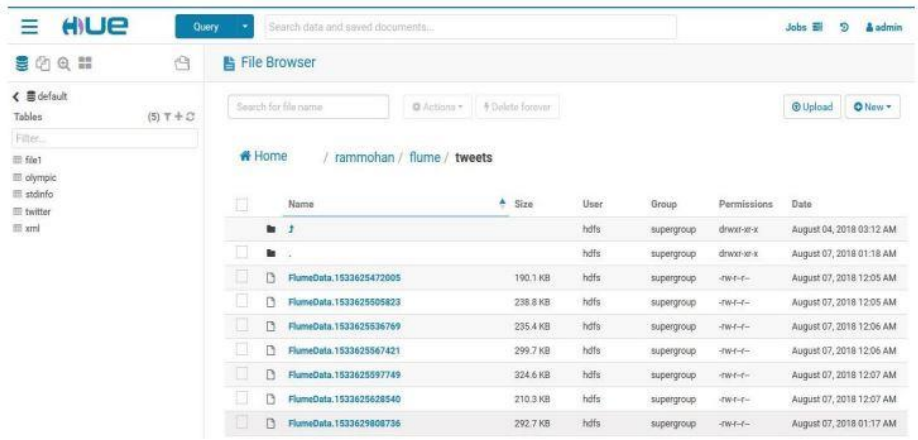
Big Data is also a data with a huge size. Big Data is a term used to describe a collection of data that is huge in size and yet growing exponentially with time. Any data which possess three characteristics i.e., volume, variety and velocity is termed to be a big data.The size of available data has been grown at an increasing rate. This applies to companies and to individuals. A text file is a few kilo bytes a sound file is a few mega bytes while a full length movie is a few giga bytes.More sources of data are added on continuous basis. For companies, in the old days, all data was generated internally by employees. Currently, the data is generated by employees, partners and customers. For a group of companies, the data is also generated by machines. For example, hundreds of millions of smart phones send a variety of information to the network infrastructure.

This data did not exist five years ago.More sources of data with a larger size of data combine to increase the volume of data that has to be analyzed. This is a major issue for those looking to put that data to use instead of letting it just disappear. Peta byte data sets are common these days and exa bytes is not far away to the server and waits for delivery of the result. That scheme works when the incoming data rate is slower than the batch processing rate and when the result is useful despite the delay. With the new sources of data such as social and mobile applications, the batch process breaks down. The data is now streaming into the server in real time, in a continuous fashion and the result is only useful if the delay is short.
From excel tables and databases, data structure has changes to lose it’s structure and to add

hundreds of formats. Pure text, photo, audio, video, web, GPS data, sensor data, relational databases, documents, SMS, pdf, flash etc. One no longer has control over the input data format. Structure can no longer be imposed like in the past in order to keep control over the analysis. As new applications are introduced new data formats come to life.

We can use Sqoop to import data from a relational database management system (RDBMS) such as MySQL or Oracle or a mainframe into the Hadoop Distributed File System (HDFS), transform the data in Hadoop MapReduce, and then export the data back into an RDBMS. In general, for each action that we want to perform with Sqoop, we will have to connect to the desired database through a JDBC connector indicating IP address, port, and access credentials. Next, we must add the parameters that will configure our request.

Flume is used in our project to bring the datasets from web server which means like twitter and keeps them in an agent, Agent contains three parts in it, and they are source, sink and channel.
Web browser brings the data and keeps in source of agent source send them to one or more channels, channel act as a bridge between source and sink. Channel sends them to sink at last stores the data to centralized hub like HDFS



Case1: Connecting sqoop to oracle log running or edgenode.
Case2: Connecting sqoop to oracle running on remote machine using jdbc url and port no.1521 of oracle.
Case3: Connecting sqoop to mysql running on remote machine and port no is 3306. Similarly we can test with any other databases

TASK 12

Aim: Integrating Hadoop with other data analytic framework like R

Procedure:

We will present three approaches to integrate R and Hadoop: R and Streaming, Rhipe and RHadoop. There are also other approaches to integrate R and Hadoop. For example RODBC/RJDBC could be used to access data from R but a survey on Internet shows that the most used approaches for linking R and Hadoop are Streaming, Rhipe (Cleveland, 2010) and RHadoop(Prajapati, 2013).

The general structure of the analytics tools integrated with Hadoop can be viewed as a layered architecture presented in figure 1.

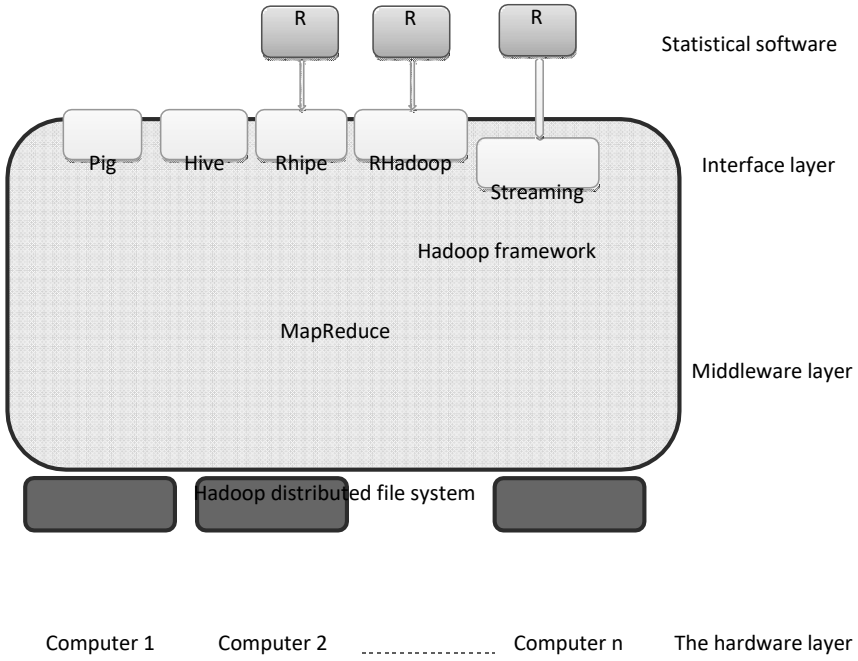
The first layer is the hardware layer – it consists in a cluster of (commodity) computers. The second layer is the middleware layer – Hadoop. It manages the distributions of the files by using HDFS and the MapReduce jobs. Then it comes a layer that provides an interface for data analysis. At thislevel we can have a tool like Pig which is a high-level platform for creating MapReduce programs using a language called Pig-Latin. We can also have Hive which is a data warehouse infrastructure developed by Apache and built on top of Hadoop. Hive provides facilities for running queries and data analysis using an SQL-like language called HiveQL and it also provides support for implementing MapReduce tasks.

Besides these two tools we can implement at this level an interface with other statistical software like R. We can use Rhipe or Rhadoop libraries that build an interface between Hadoop and R, allowing users to access data from the Hadoop file system and write their own scripts for implementing Map and Reduce jobs, or we can use Streaming that is a technology integratedin Hadoop.

Hadoop can be also integrated with other statistical software like SASor SPSS.

Hadoop and data analysis tools

Figure 1



We will analyze these options of integration between R and Hadoop from different point of views: licensing, complexity of installation, benefits and limitations.

R AND STREAMING

Streaming is a technology integrated in the Hadoop distribution that allows users to run Map/Reduce jobs with any script or executable that reads data from standard input and writes the results to standard output as the mapper or reducer. This means that we can use Streaming together with R scripts in the map and/or reduce phase since R can read/write data from/to standard

input. In this approach there is no client-side integration with R because the user will use the Hadoop command line to launch the Streaming jobs with the arguments specifying the mapper and reducer R scripts.

A command line with map and reduce tasks implemented as R scripts would look like this:

An example of a map-reduce task with R and Hadoop integrated by Streaming framework

Figure 2

```
$ ${HADOOP_HOME}/bin/Hadoop jar
${HADOOP_HOME}/contrib/streaming/*.jar \
-inputformat org.apache.hadoop.mapred.TextInputFormat \
-input input_data.txt \
-output output \
-mapper /home/tst/src/map.R \
-reducer /home/tst/src/reduce.R \
-file /home/tst/src/map.R \
-file /home/tst/src/reduce.R
```

Here we supposed that the data in the file "input_data.txt" was already copied from the local file system to the HDFS. The meaning of the commandline parameters are:

"-inputformat org.apache.hadoop.mapred.TextInputFormat" specifies the input format for the job (we stored our input data in a text file);

"-input input_data.txt" specifies the input data file of our job; "-output output" sets the output directory of the job;

“-mapper /home/tst/src/map.R” specifies the map phase executable. In this example we used an R script named map.Rlocated in /home/tst/src/ directory;

“-reducer /home/tst/src/reduce.R” specifies the reduce phase executable. In our example, the reducer is also an R script named reduce.Rlocated in /home/tst/src/ directory;

“-file /home/tst/src/map.R” indicates that the R script map.R should be copied to the distributed cache, made available to the map tasks and causes the map.R script to be transferred to the cluster machines where the map- reduce job will be run;

“-file /home/tst/src/reduce.R” indicates that the R script reduce.R should be copied to the distributed cache an made available to the map tasks and causes the reduce.R script to be transffered to the cluster machines wherethe map-reduce job will be run.

The integration of R and Hadoop using Streaming is an easy task because the user only needs to run Hadoop command line to launch the Streaming job specifying the mapper and reducer scripts as command line arguments. This approach requires that R should be installed on every DataNode of the Hadoop cluster but this is simple task.

The licensing scheme need for this approach implies an Apache 2.0 license for Hadoop and a combination of GPL-2 and GPL-3 for R.

RHIPE

Rhipe stands for “Rand Hadoop Integrated Programming Environment”and is an open source project that provides a tight integration between R and Hadoop. It allows the user to carry out data analysis of big data directly in R, providing R users the same facilities of Hadoop as Java developers have. The software package is freely available for download at www.datadr.org.

The installation of the Rhipe is somehow a difficult task. On each DataNode the user should install R, Protocol Buffers and Rhipe and this is not an easy task: it requires that R should be built as a shared library on each node, the Google Protocol Buffers to be built and installed on each node and to install the Rhipe itself. The Protocol Buffers are needed for data serialization, increasing the efficiency and providing interoperability with other languages.

The Rhipe is an R library which allows running a MapReduce job within R. The user should write specific native R map and reduce functions and Rhipe will manage the rest: it will transfer them and invoke them from map and reduce tasks. The map and reduce inputs are transferred using a Protocol Buffer encoding scheme to a Rhipe C library which uses R to call the map and reduce functions. The advantages of using Rhipe and not the parallel R packages consist in its integration with Hadoop that provides a data distribution scheme using Hadoop distributed file system across a cluster of computers that tries to optimize the processor usage and provides fault tolerance.

The general structure of an R script that uses Rhipe is shown in figure 3 and one can easily note that writing such a script is very simple.

The structure of an R script using Rhipe

Figure 3

```
library(Rhipe)

rhinit(TRUE, TRUE);

map<-expression ( {lapply(map.values,function(mapper)…)})

reduce<-expression(5 pre = {…},
```

```
6 reduce = {…},7 post = {…},
8 )

x <- rhmr(
map=map, reduce=reduce,
ifolder=inputPath,
ofolder=outputPath,
inout=c('text', 'text'),
jobname='a job name'))

rhex(z)
```

The script should begin with loading the Rhipe library into memory (line 1) and initializing the Rhipe library (line 2). Line 3 defines the map expression to be executed by the Map task. Lines 4 to 8 defines the reduce expression. Lines 4 to 8 define the reduce expression consisting in three callbacks. The pre block of instructions (line 5) is called for each unique map output key before these values being sent to the reduce block. The reduceblock (line 6) is called then with a vector of values as argument and in the end, the postblock (line 7) is called to emit the output (key, value) pair. Line 9 shows the call of rhmrfunction which set up the job (creates a MapReduceobject) and rhexfunction call (line 15) that launches the MapReduce job to the Hadoop framework. Rhipe also provides functions to communicate with Hadoop during the MapReduce process like rhcollect that allows writing data to Hadoop MapReduce or rhstatus that returns the status of the a job. Rhipe let the user to focus on data processing algorithms and the difficulties of distributing data and computations across a cluster of computers are handled by the Rhipe and library and Hadoop.

The licensing scheme needed for this approach implies an Apache 2.0 license for Hadoop and Rhipe and a combination of GPL-2 and GPL-3 for R.

RHADOOP

RHadoop is an open source project developed by Revolution Analytics (http://www.revolutionanalytics.com/) that provides client-side integration of R and Hadoop. It allows running a MapReduce jobs within R just like Rhipe and consist in a collection of four R packages:

plyrmr- plyr-like data processing for structured data, providing

common data manipulation operations on very large data setsmanaged by Hadoop;

rmr – a collection of functions providing and integration of R and MapReduce model of computation;

rdfs – an interface between R and HDFS, providing filemanagement operations within R;

rhbase- an interface between R and HBase providing database management functions for HBase within R;

Setting up RHadoop is not a complicated task although RHadoop has dependencies on other R packages. Working with RHadoop implies to install R and RHadoop packages with dependencies on each Data node of the Hadoop cluster. RHadoop has a wrapper R script called from Streaming that calls user defined map and reduce R functions. RHadoop works similarly to Rhipe allowing user to define the map and reduce operation. A script that uses RHadoop looks like:

The structure of an R script using RHadoop library(rmr)

```
map<-function(k,v) { …}

reduce<-function(k,vv) { …}

mapreduce( input =”data.txt”,

output=”output”,

map = map, reduce=reduce )

textinputformat =rawtextinp
```

First, the rmr library is loaded into memory (line 1) and then follows the definition of the map function which receives a (key,value) pair as input. The reduce function (line 3) is called with a key and a list of values as arguments for each unique map key. Finally, the scripts sets up and run the mapreduce job (line 4). It should be noted that rmr makes the client-side R environment available for map and reduce functions. The licensing scheme needed for this approach implies an Apache 2.0 license for Hadoop and RHadoop and a combination of GPL-2 and GPL-3 for R.

Official statistics is increasingly considering big data for building new statistics because its potential to produce more relevant and timely statistics than traditional data sources. One of the software tools successfully used for storage and processing of big data sets on clusters of commodity hardware is Hadoop. In this paper we presented three ways of integrating R and Hadoop for processing large scale data sets: R and Streaming, Rhipe and RHadoop. Wehave to mention that there are also other ways of integrating them like ROBDC, RJBDC or Rhive but they have some limitations. Each of the approaches presented here has benefits and limitations. While using R with Streaming raises no problems regarding installation, Rhipe and RHadoop requires some effort in order to set up the cluster. The integration with R from the client sidepart is high for Rhipe and Rhadoop and is missing for R and Streaming. Rhipeand RHadoop allows users to define and call their own map and reducefunctions within R while Streaming uses a command line approach where themapand reduce functions are passed as arguments. Regarding the licensing scheme, all three approaches require GPL-2 and GPL-3 for R and Apache 2.0 for Hadoop, Streaming, Rhipe and RHadoop.

We have to mention that there are other alternatives for large scale data analysis: Apache Mahout, Apache Hive, commercial versions of R provided by Revolution Analytics, Segue framework or ORCH, an Oracle connector for R but Hadoop with R seems to be the most used approach. For simple Map-Reduce jobs the straightforward solution is Streaming but this solution islimited to text only input data files. For more complex jobs the solution should be Rhipe or RHadoop.