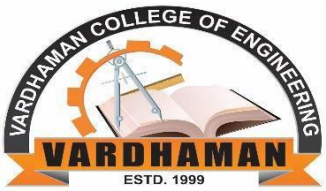
VARDHAMAN COLLEGE OF ENGINEERING, HYDERABAD

(Autonomous)

NAAC-A Grade, NBA Accredited: Affiliated to JNTUH, Hyderabad

ISO 9001:2008 Certified.

DEPARTMENT OF INFORMATION TECHNOLOGY



Laboratory Manual
of
BIG DATA ANALYTICS

(III B.Tech- VI SEMESTER)

(VCE-R18)

<table>
<tr><td>Course Title</td><td colspan="4">BIG DATA ANALYTICS</td><td colspan="2">Course Type</td><td colspan="2">Integrated</td></tr>
<tr><td>Course Code</td><td>A4513</td><td>Credits</td><td colspan="2">2</td><td colspan="2">Class</td><td colspan="2">IIIRD YEAR II SEM</td></tr>
<tr><td rowspan="5">Course Structure</td><td>TLP</td><td>Credits</td><td>Contact Hours</td><td>Work Load</td><td colspan="2" rowspan="2">Total Number of Classes Per Semester</td><td colspan="2" rowspan="2">Assessment in Weightage</td></tr>
<tr><td>Theory</td><td>3</td><td>3</td><td>3</td></tr>
<tr><td>Tutorial</td><td>0</td><td>0</td><td>0</td><td rowspan="2">Theory</td><td rowspan="2">Practical</td><td rowspan="2">CIE</td><td rowspan="2">SEE</td></tr>
<tr><td>Practice</td><td>2</td><td>2</td><td>1</td></tr>
<tr><td>Total</td><td>5</td><td>5</td><td>4</td><td>42</td><td>28</td><td>30</td><td>70</td></tr>
<tr><td rowspan="3">Course Instructors</td><td colspan="8">Course Lead: R ARUN KUMAR, G SRAVAN KUMAR, ASSISTANT PROF, IT, VCE</td></tr>
<tr><td colspan="4">Theory</td><td colspan="4">Practice</td></tr>
<tr><td colspan="4">G SRAVAN KUMAR
R ARUN KUMAR</td><td colspan="4">G SRAVAN KUMAR
R ARUN KUMAR</td></tr>
</table>

COURSE OVERVIEW

Big data analytics is the use of advanced analytic techniques against very large, diverse data sets that include structured, semi-structured and unstructured data, from different sources, and in different sizes from terabytes to zetta bytes. Big data is a term applied to data sets whose size or type is beyond the ability of traditional relational databases to capture, manage and process the data with low latency. Analysis of big data allows users to make better and faster decisions using data that was previously inaccessible or unusable.

COURSE OBJECTIVE

The course enables the learner to warehouse and analyze the data efficiently using modern tools like Hadoop, Hive, MongoDB and Dataframes.

COURSE OUTCOMES (COs)

After the completion of the course, the student will be able to:

CO#	Course Outcomes	POs	PSOs
A4513.1	Understand the fundamental concepts of big data analytics.	-	-
A4513.2	Apply various Technologies to meet Challenges in Big Data analytics	5	1
A4513.3	Apply the HADOOP and Map Reduce technologies associated with big data analytics.	5	1
A4513.4	Apply MangoDB on Unstructured data	4,5	1
A4513.5	Analyze Big Data applications Using Pig and Hive.	4,5	1

BLOOM'S LEVEL OF THE COURSE OUTCOMES

CO#	Bloom's Level					
	Remember (L1)	Understand (L2)	Apply (L3)	Analyze (L4)	Evaluate (L5)	Create (L6)
A3555.1		√				
A3555.2			√			
A3555.3			√			
A3555.4			√			
A3555.5				√		

.....

COURSE ARTICULATION MATRIX

CO#/ Pos	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
A3502.2					2								2	
A3502.3					3								3	
A3502.4				2	3								3	
A3502.5				2	3								3	

Note: 1-Low, 2-Medium, 3-High

COURSE ASSESSMENT

S No	Component		Duration in Hours	Component Wise Marks	Total Marks	Weightage	Marks
1	Continuous Internal Evaluation (CIE)	Theory: Test-1	1.5	20	100	0.3	30
2		Theory: Test-2	1.5	20			
3		Alternate Assessment*	-	20			
4		Practical Exam	2	40			
5	Semester End Exam (SEE)		3	100	100	0.7	70
						Total Marks	**100**

LIST OF PROGRAMS FORPRACTICE:

No	Title of the Experiment	Tools and Techniques	Expected Skill /Ability
1	**Week-1:** Installation Hadoop	CLoudera	Installation of hadoop
1.	**Week-2:** a) Basic file commands b) Web Based User Interface	OS: Linux /Windows , IDE: Cloud Era Modern Tool:Hadoop/mongodb	Apply basic hadoop commands
2.	**Week-3:** Reading & Writing to files		Apply basic hadoop-file commands.

No	Title of the Experiment	Tools and Techniques	Expected Skill /Ability
3.	**Week-4:** Run a word count program		Apply Hadoop mapreduce/mongodb
4.	**Week-5:** Incremental mapreduce		
5.	**Week-6:** Indexing and Aggregation		Mongodb
6.	**Week-6:** Pig Queries		Apply Pig-Programming to solve the quires
7.	**Week-7:**Hive Tables (Managed Tables and External Tables)	OS: Linux /Windows , IDE: Cloud Era Modern Tool:HIVE	Apply HIVE program
8.	**Week-8:**Hive Tables		Apply Hive Programming
9.	**Week-9: MapReduce weather dataset**		Apply HADOOP Programming
10.	**Week-10:** **MapReduce weather dataset -PIG**	OS: Linux /Windows , IDE: Hadoop Modern Tool: HADOOP OS: Linux /Windows , IDE: HADOOP Modern Tool: HADOOP	Apply HADOOP to import and query data processing

LAB MANUAL CONTENTS

This manual includes five Parts.

PART	CONTENT
1	HADOOP ESSENTIALS
2	WEEKLY LAB EXCERCISES
3	ONLINE RESOURCES
4	POSSIBLE VIVA QUESTIONS
5	KNOWLEDGE BASE

PART-1

HADOOP ESSENTIALS

HADOOP ESSENTIALS

This Manual provides the basic Categorization of Cloud basics to work on scenario based learning on different Ecosystems.

1 SOFTWARE STUDY -HADOOP AND ECOSYSTEM OF HADOOP:

1.1 OBJECTIVE

How to Manage the Hadoop ecosystem

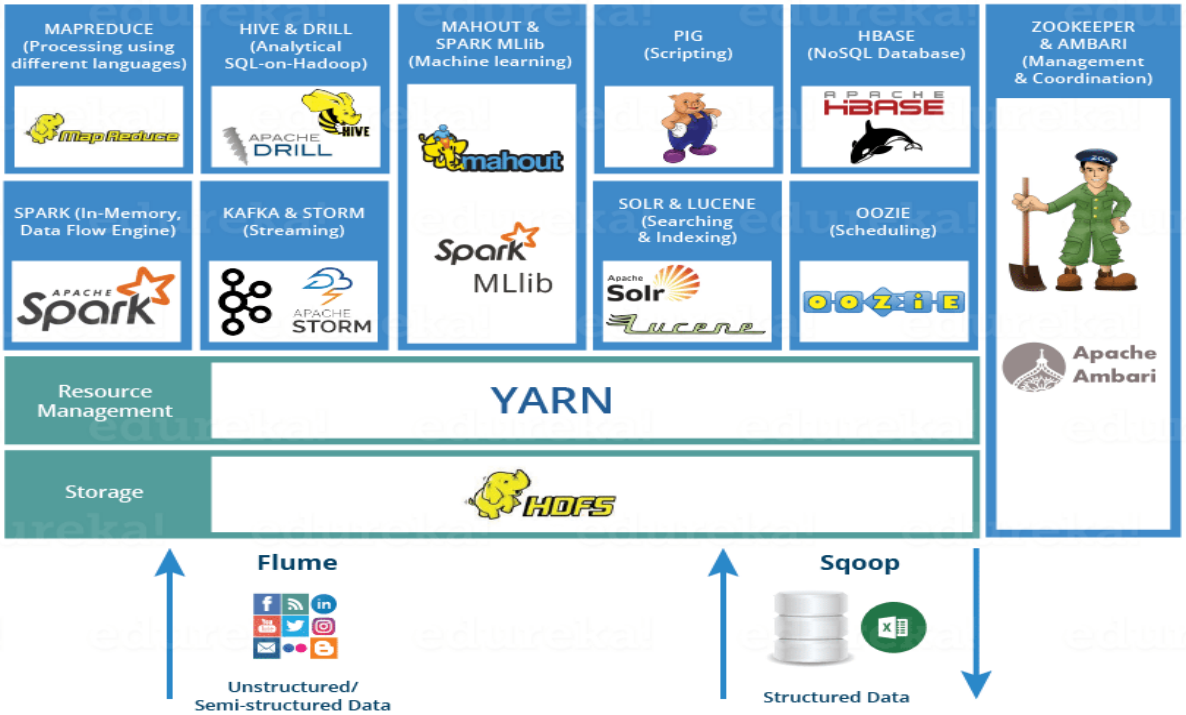
1.2 PROCEDURE

Source : Edureka(blog/hadoop-ecosystem) https://www.edureka.co/blog/hadoop-ecosystem

Hadoop Ecosystem is neither a programming language nor a service, it is a platform or framework which solves big data problems. You can consider it as a suite which encompasses a number of services (ingesting, storing, analyzing and maintaining) inside it. Let us discuss and get a brief idea about how the services work individually and in collaboration.

Below are the Hadoop components, that together form a Hadoop ecosystem, I will be covering each of them in this blog:

- **HDFS** -> *Hadoop Distributed File System*
- **YARN** -> *Yet Another Resource Negotiator*
- **MapReduce** -> *Data processing using programming*
- **Spark** -> In-memory Data Processing
- **PIG**, **HIVE**-> *Data Processing Services using Query (SQL-like)*
- **HBase** -> *NoSQL Database*
- **Mahout**, **Spark MLlib** -> *Machine Learning*
- **Apache Drill** -> *SQL on Hadoop*
- **Zookeeper** -> *ManagingCluster*
- **Oozie** -> *Job Scheduling*
- **Flume**, **Sqoop** -> *Data Ingesting Services*
- **Solr & Lucene** -> *Searching & Indexing*
- **Ambari** -> *Provision, Monitor and Maintain cluster*



HDFS

- *Hadoop Distributed File System* is the core component or you can say, the backbone of Hadoop Ecosystem.
- HDFS is the one, which makes it possible to store different types of large data sets (i.e. structured, unstructured and semi structured data).
- HDFS creates a level of abstraction over the resources, from where we can see the whole HDFS as a single unit.
- It helps us in storing our data across various nodes and maintaining the log file about the stored data (metadata).
- HDFS has two core components, i.e. **NameNode and DataNode**.
    1. The **NameNode** is the main node and it doesn't store the actual data. It contains metadata, just like a log file or you can say as a table of content. Therefore, it requires less storage and high computational resources.
    2. On the other hand, all your data is stored on the **DataNodes** and hence it requires more storage resources. These DataNodes are commodity hardware (like your laptops and desktops) in the distributed environment. That's the reason, why Hadoop solutions are very cost effective.
    3. You always communicate to the NameNode while writing the data. Then, it internally sends a request to the client to store and replicate data on various DataNodes.

YARN

Consider YARN as the brain of your Hadoop Ecosystem. It performs all your processing activities by allocating resources and scheduling tasks.

- It has two major components, i.e. **ResourceManager and NodeManager**.
    1. **ResourceManager** is again a main node in the processing department.
    2. It receives the processing requests, and then passes the parts of requests to corresponding NodeManagers accordingly, where the actual processing takes place.
    3. **NodeManagers** are installed on every DataNode. It is responsible for execution of task on every single DataNode.

-

MAPREDUCE

It is the core component of processing in a Hadoop Ecosystem as it provides the logic of processing. In other words, MapReduce is a software framework which helps in writing applications that processes large data sets using distributed and parallel algorithms inside Hadoop environment.

- In a MapReduce program, **Map() and Reduce()** are two functions.
    1. The **Map function** performs actions like filtering, grouping and sorting.
    2. While **Reduce function** aggregates and summarizes the result produced by map function.
    3. The result generated by the Map function is a key value pair (K, V) which acts as the input for Reduce function.

APACHE PIG

- PIG has two parts: **Pig Latin**, the language and **the pig runtime,** for the execution environment. You can better understand it as Java and JVM.
- It supports *pig latin* language, which has SQL like command structure.

As everyone does not belong from a programming background. So, Apache PIG relieves them. *You might be curious to know how?*

Well, I will tell you an interesting fact:

10 line of pig latin = approx. 200 lines of Map-Reduce Java code

- It gives you a platform for building data flow for ETL (Extract, Transform and Load), processing and analyzing huge data sets.

APACHE HIVE

- Facebook created HIVE for people who are fluent with SQL. Thus, HIVE makes them feel at home while working in a Hadoop Ecosystem.
- Basically, HIVE is a data warehousing component which performs reading, writing and managing large data sets in a distributed environment using SQL-like interface.

HIVE + SQL = HQL

- The query language of Hive is called Hive Query Language(HQL), which is very similar like SQL.
- It has 2 basic components: **Hive Command Line and JDBC/ODBC driver**.
- The **Hive Command line** interface is used to execute HQL commands.
- While, Java Database Connectivity (**JDBC**) and Object Database Connectivity (**ODBC**) is used to establish connection from data storage.

APACHE MAHOUT

Now, let us talk about Mahout which is renowned for machine learning. Mahout provides an environment for creating machine learning applications which are scalable.

APACHE SPARK

- Apache Spark is a framework for real time data analytics in a distributed computing environment.
- The Spark is written in Scala and was originally developed at the University of California, Berkeley.
- It executes in-memory computations to increase speed of data processing over Map-Reduce.

APACHE HBASE

- HBase is an open source, non-relational distributed database. In other words, it is a NoSQL database.
- It supports all types of data and that is why, it's capable of handling anything and everything inside a Hadoop ecosystem.
- It is modelled after Google's BigTable, which is a distributed storage system designed to cope up with large data sets.
- The HBase was designed to run on top of HDFS and provides BigTable like capabilities.
- It gives us a fault tolerant way of storing sparse data, which is common in most Big Data use cases.

APACHE DRILL

As the name suggests, Apache Drill is used to drill into any kind of data. It's an open source application which works with distributed environment to analyze large data sets.

- It is a replica of Google Dremel.
- It supports different kinds NoSQL databases and file systems, which is a powerful feature of Drill.

APACHE ZOOKEEPER

- Apache Zookeeper is the coordinator of any Hadoop job which includes a combination of various services in a Hadoop Ecosystem.
- Apache Zookeeper coordinates with various services in a distributed environment.

APACHE OOZIE

Consider Apache Oozie as a clock and alarm service inside Hadoop Ecosystem. For Apache jobs, Oozie has been just like a scheduler. It schedules Hadoop jobs and binds them together as one logical work.

APACHE SQOOPand FLUME

Now, let us talk about another data ingesting service i.e. Sqoop. The major difference between Flume and Sqoop is that:

- Flume only ingests unstructured data or semi-structured data into HDFS.
- While Sqoop can import as well as export structured data from RDBMS or Enterprise data warehouses to HDFS or vice versa.

APACHE SOLR & LUCENE

- Apache Lucene is based on Java, which also helps in spell checking.
- If Apache Lucene is the engine, Apache Solr is the car built around it. Solr is a complete application built around Lucene.

APACHE AMBARI

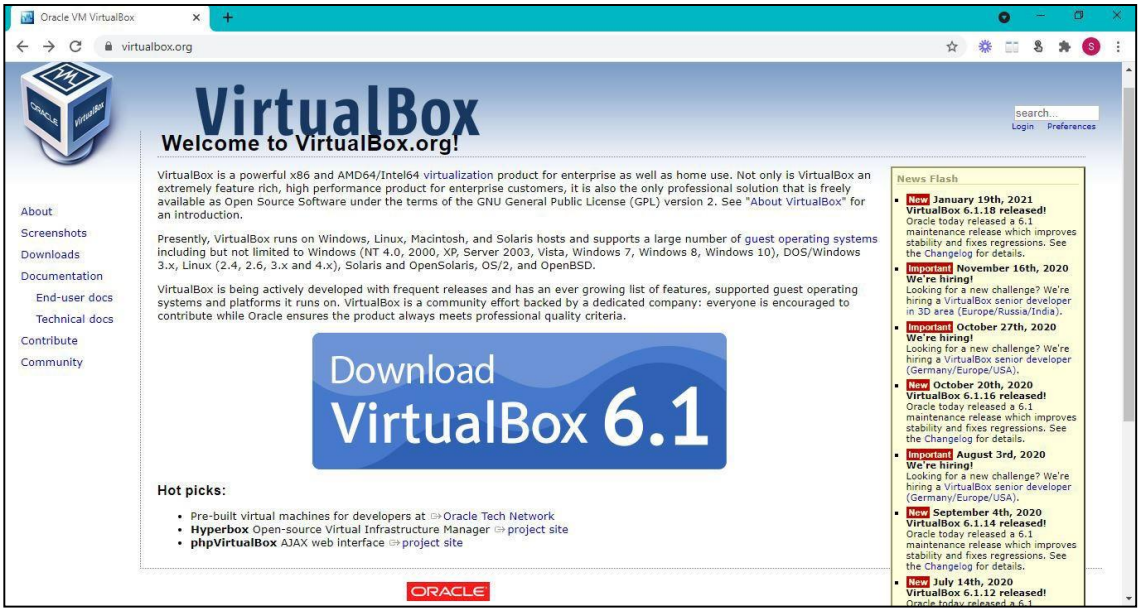
Ambari is an Apache Software Foundation Project which aims at making Hadoop ecosystem more manageable.

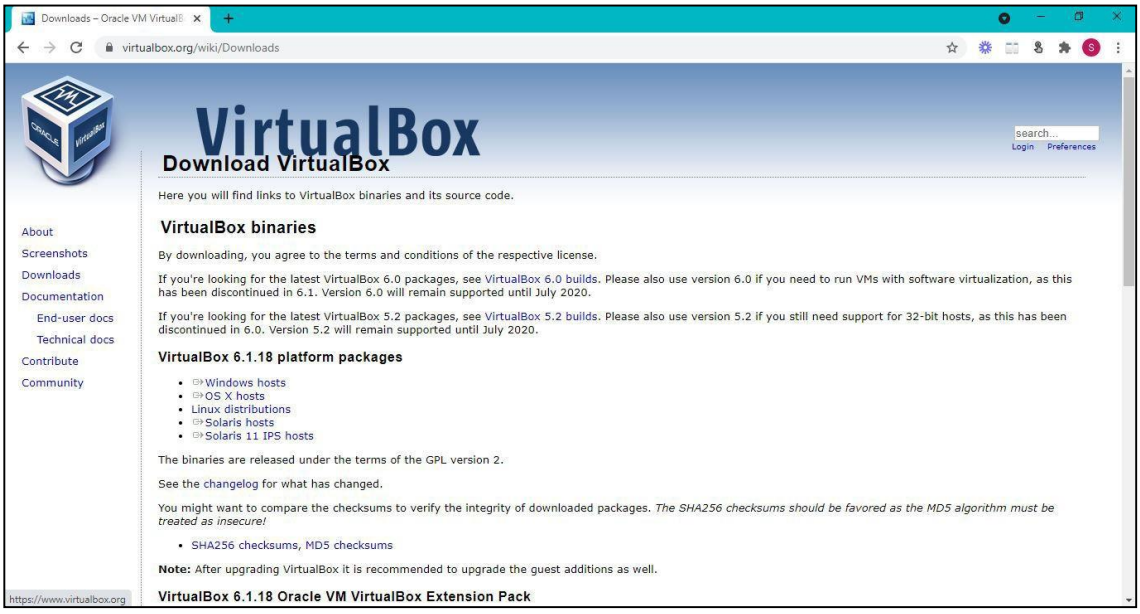
PART-2

WEEKLY LAB EXCERCISES

Week1: Perform setting up and installing Hadoop

Step1: Visit www.virtualbox.org, look for Downloads to view VirtualBox platform package of Windows hosts and click on it to download.

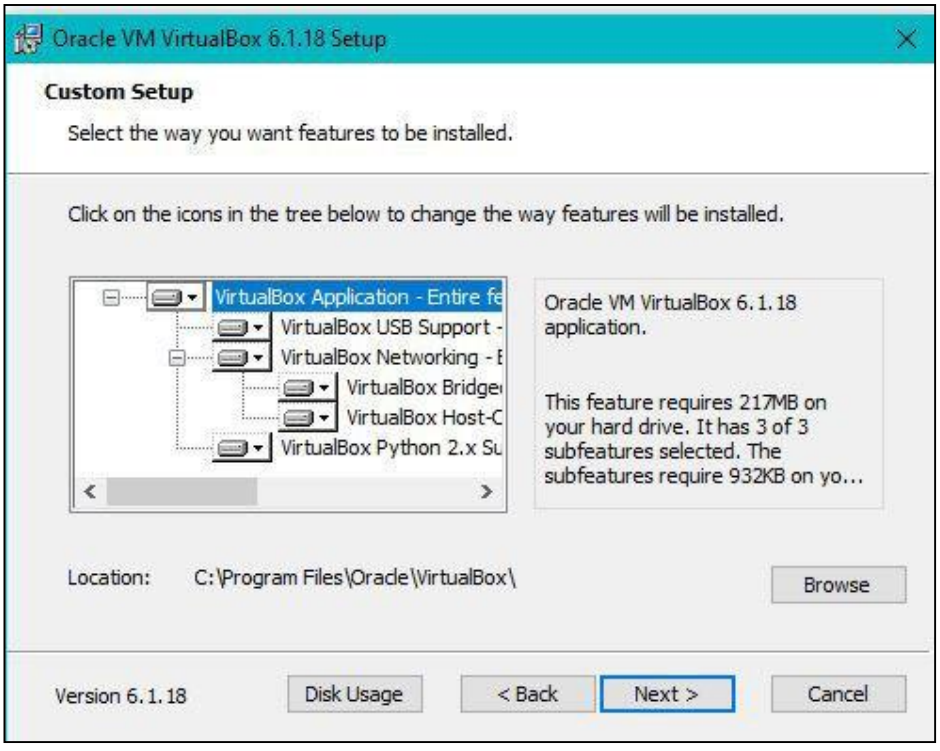




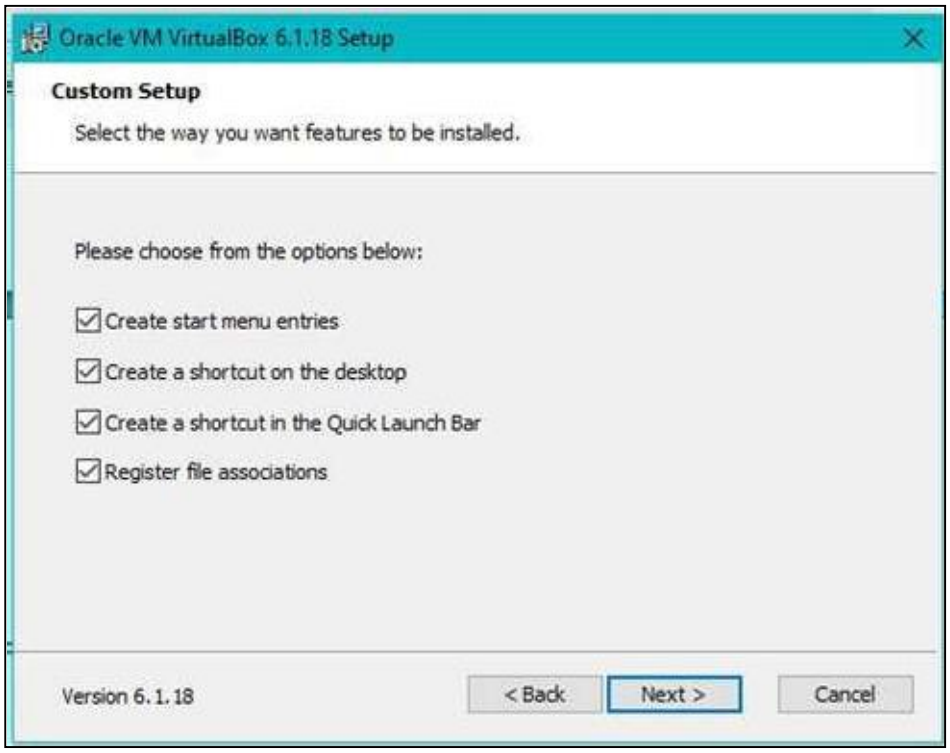
Step2:

a) Double click on ***VirtualBox-6.1.18-142142-Win*** to install and the following screen. Click on Next

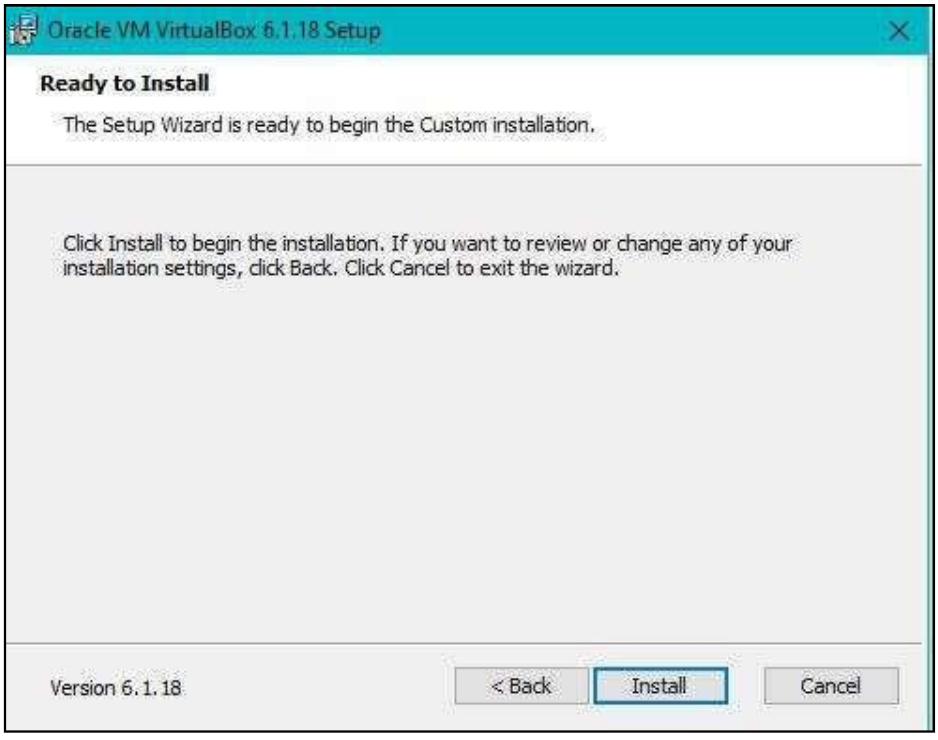
b) With the previous clicks, the following appears screen on which click on Next



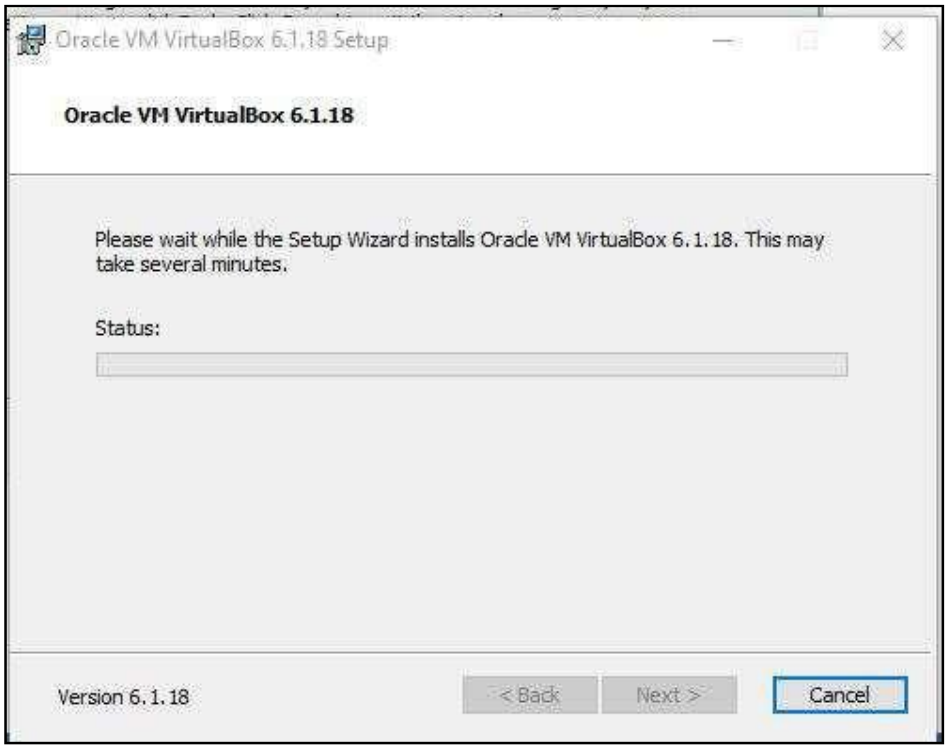
c) Check all options and click Next



d) Click Install

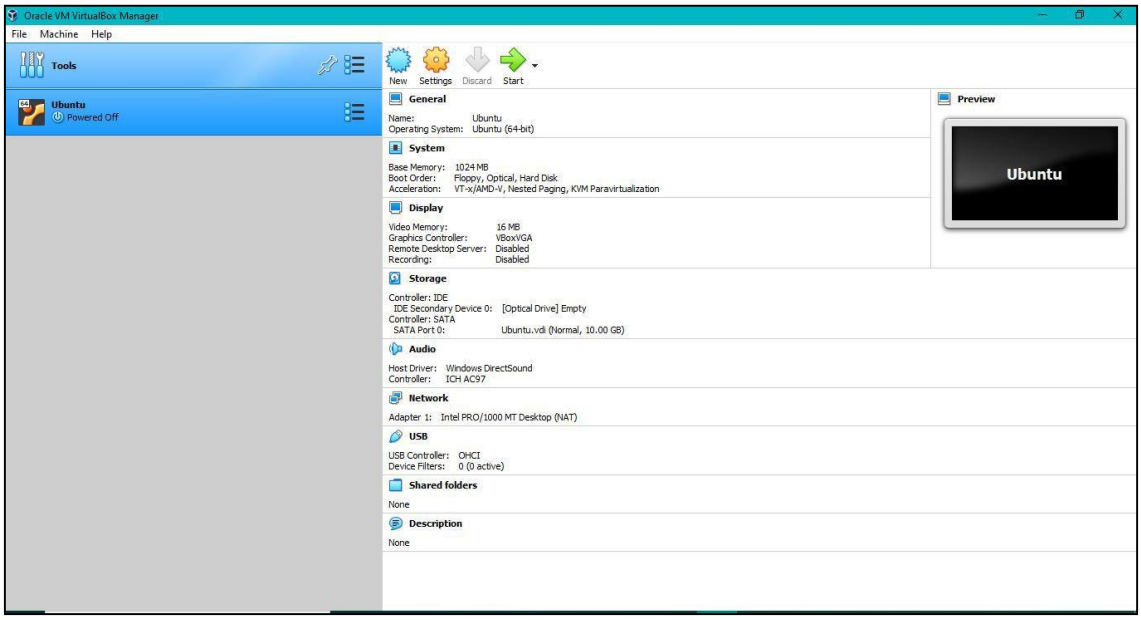


e) Virtual box is being



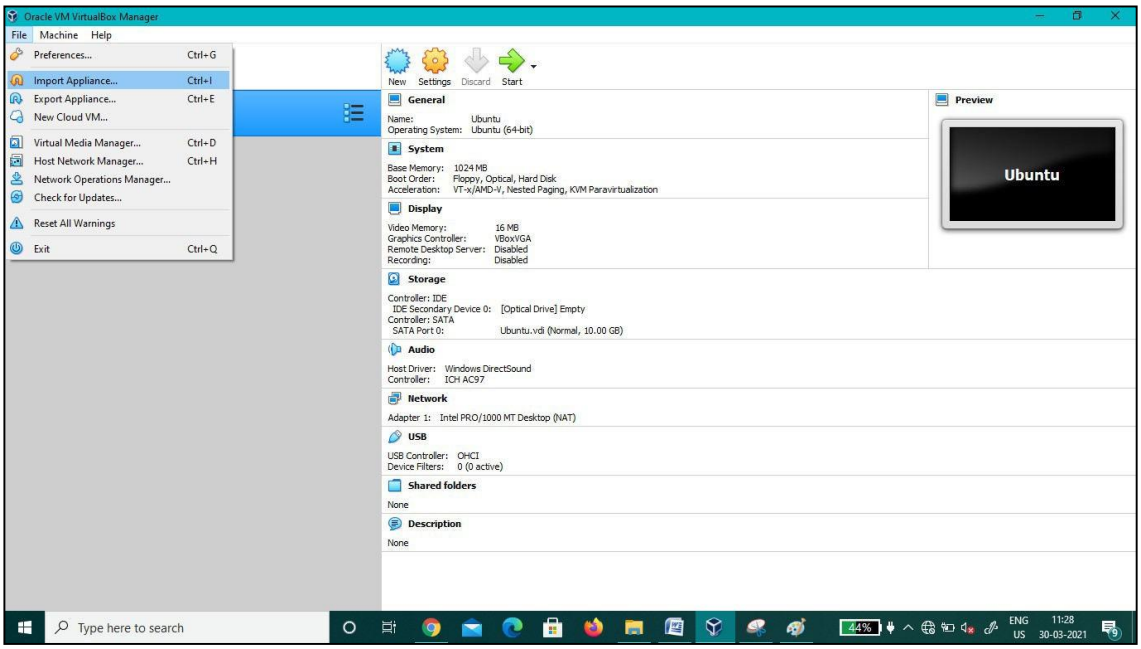
f) VirtualBox has installed and check option to start Oracle VM VirtualBox 6.1.18

g) Oracle VM VirtualBox is being run.

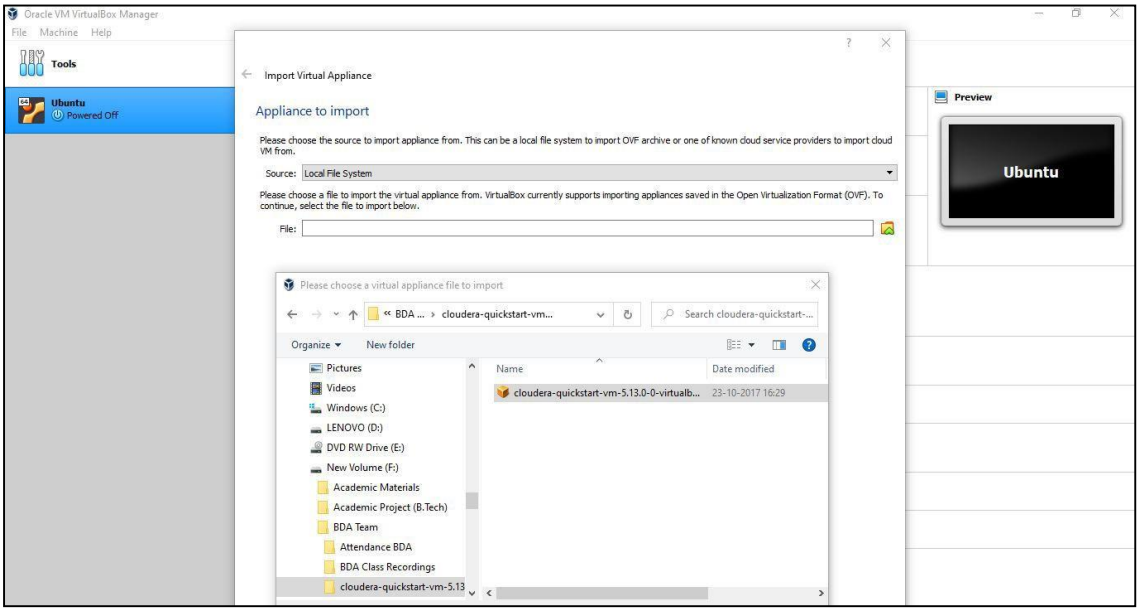


Step 3:

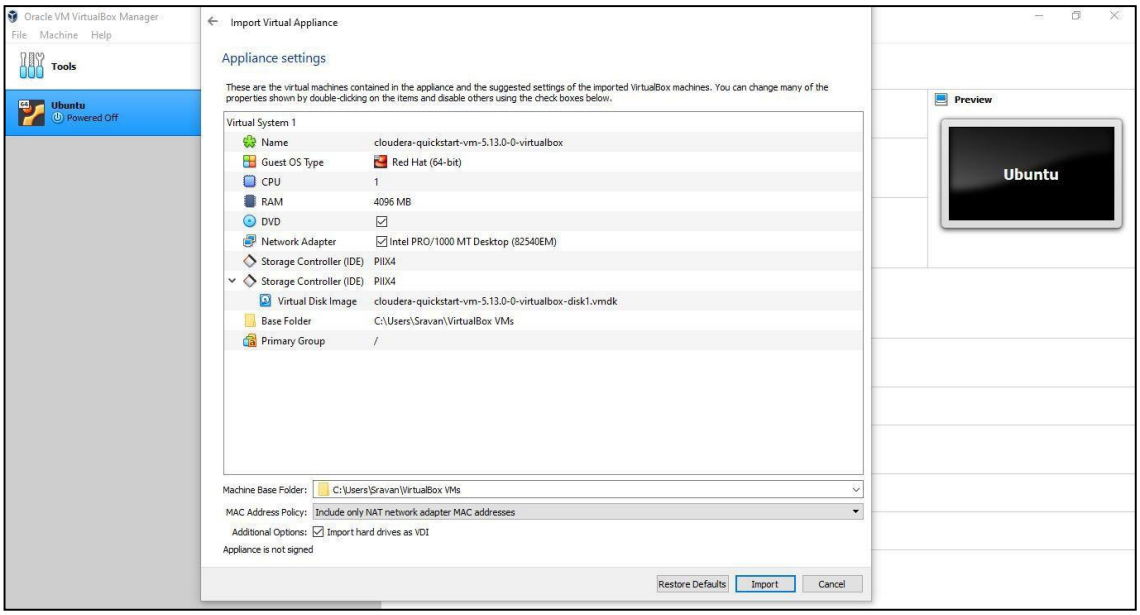
a) Download *cloudera-quickstart-vm-5.13.0-0-virtualbox* and click File -> ImportAppliance



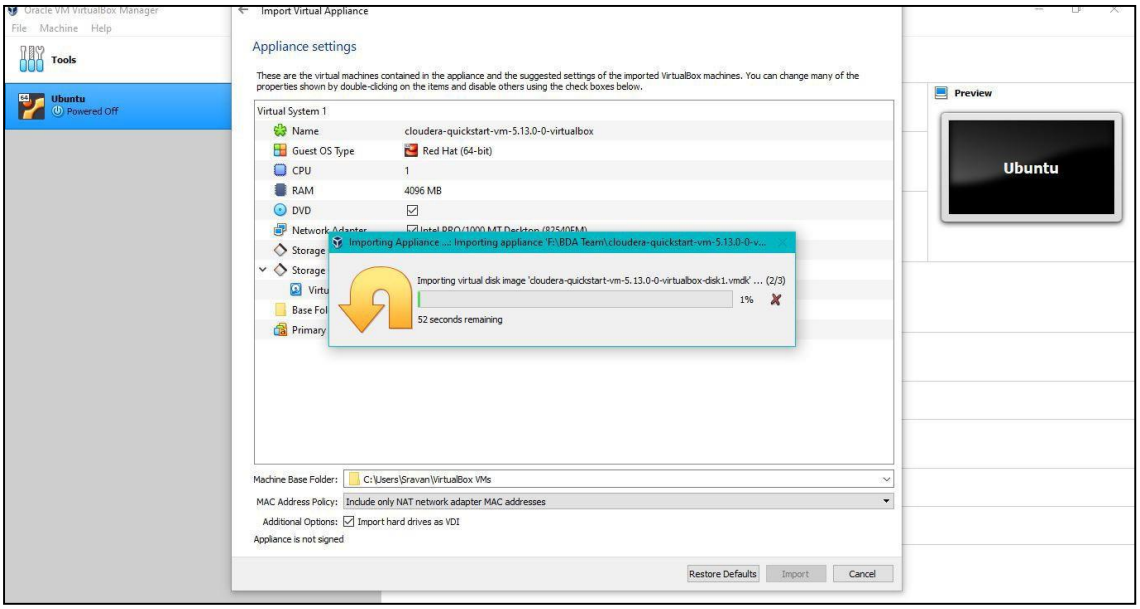
b) Once Import Appliance is clicked, it prompts to browse for cloudera-quickstart-vm-5.13.0-0- virtualbox (Open Virtualization Format) and click Open -> Next



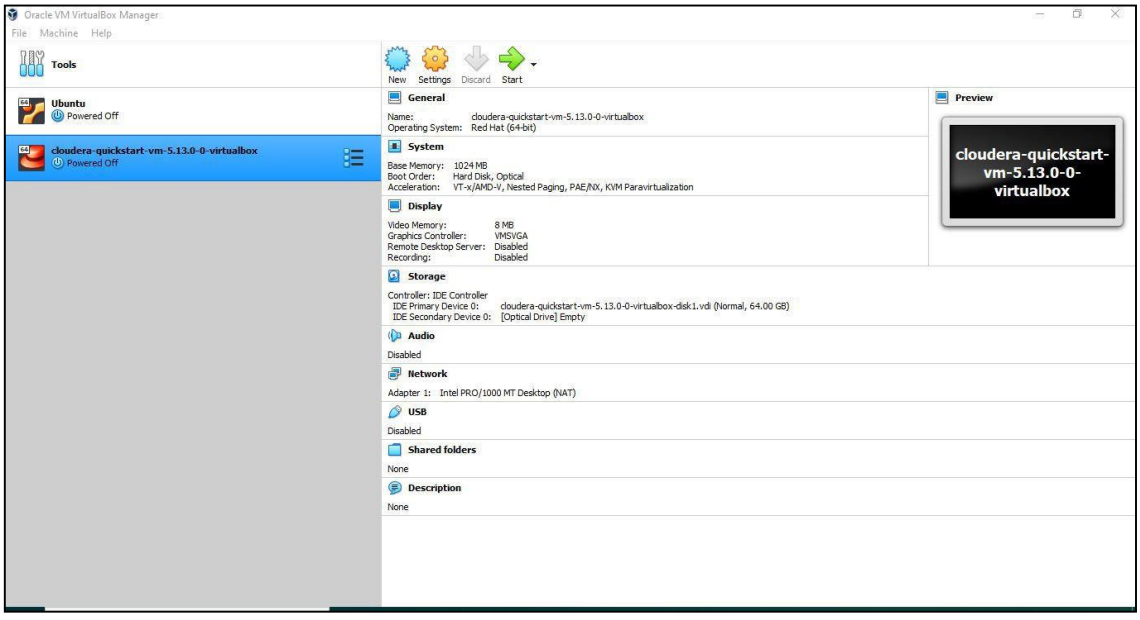
c) It prompts Appliance settings where configuration is made. Click Import



d) cloudera-quickstart-vm-5.13.0-0 is being imported into VirtualBox



e) cloudera-quickstart-vm-5.13.0-0 is imported and click Start (Green Coloured Arrow) to launch Cloudera Environment.

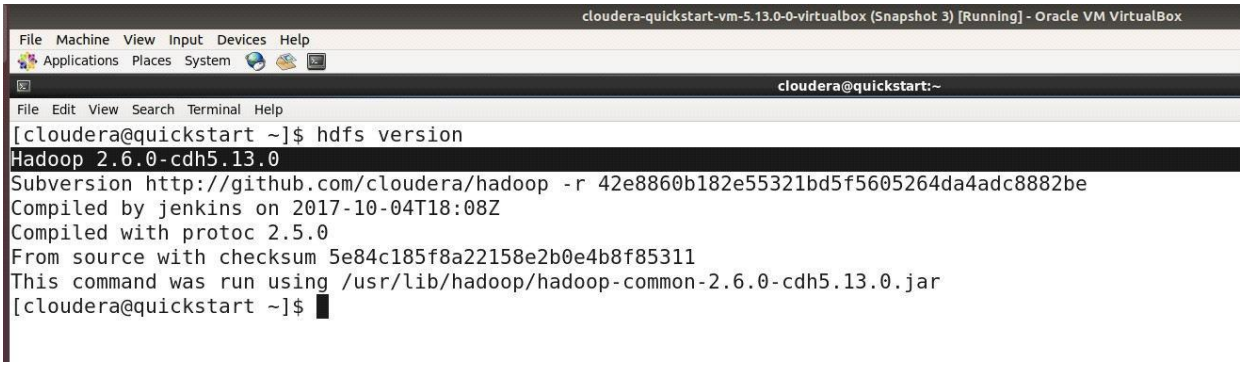


Week- 2 & 3: Basic HDFS Commands: ls, copy, get put etc., & File Management tasks in Hadoop

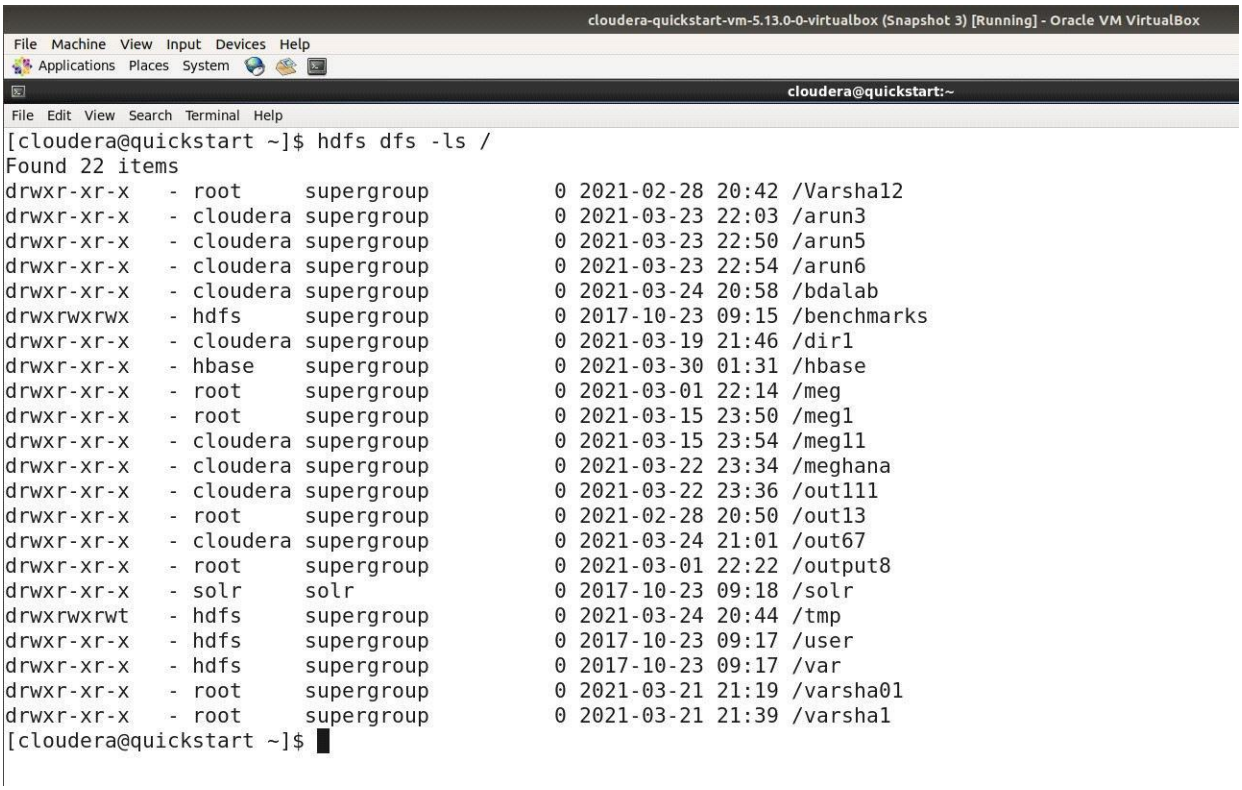
a) Adding files and directories b) Retrieving files and c) Deleting files.

The following HDFS commands are used to access the Hadoop File System (HDFS).

1. version: This command is used to access the version of HDFS.

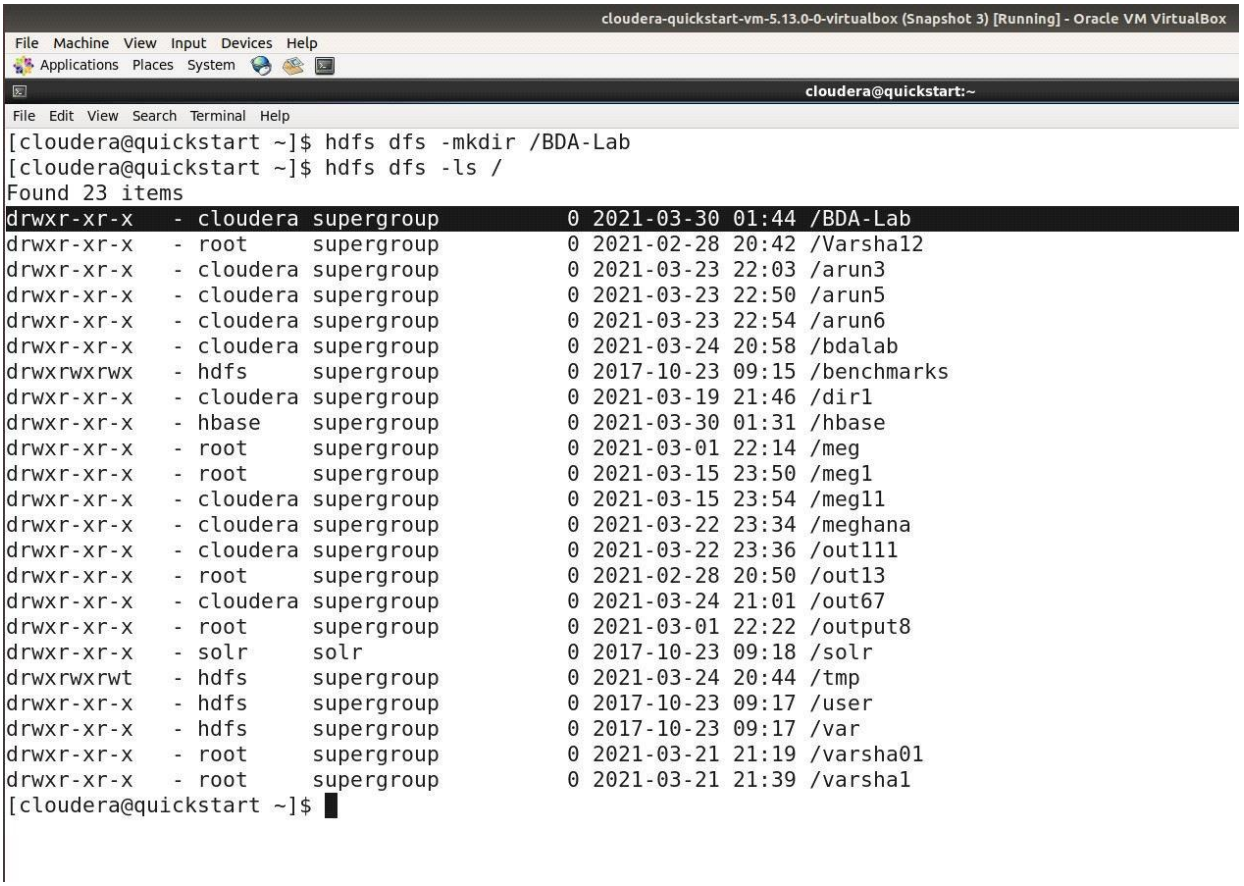


2. ls: This command is used to access the list of files and directories in HDFS.

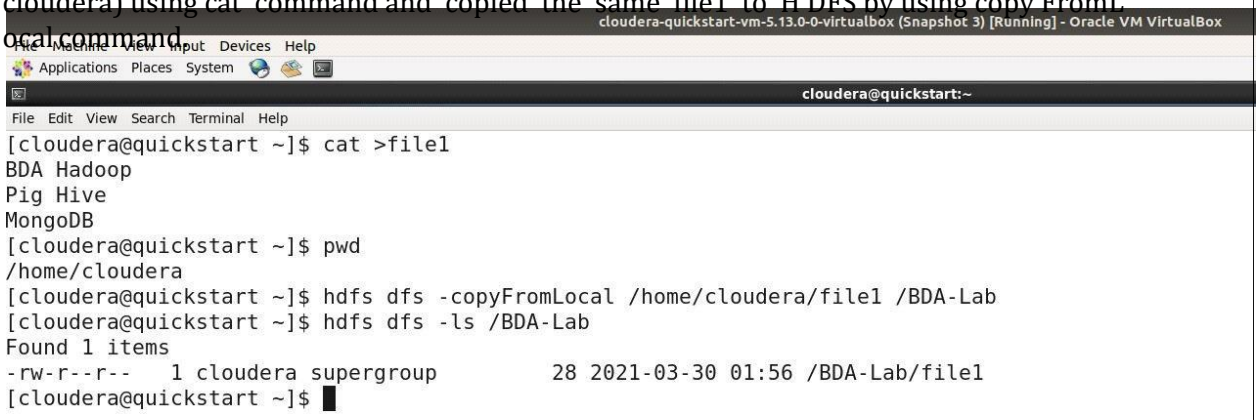


3. mkdir: T his command is used to the directory in file sy stem.

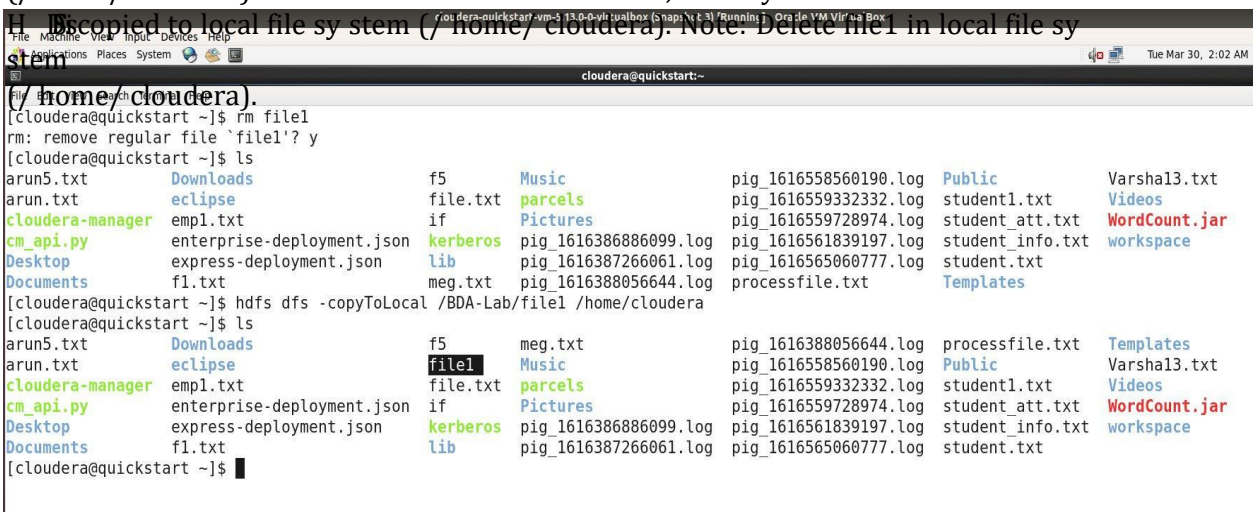
T he following command is used to check whether the directory is created or not.



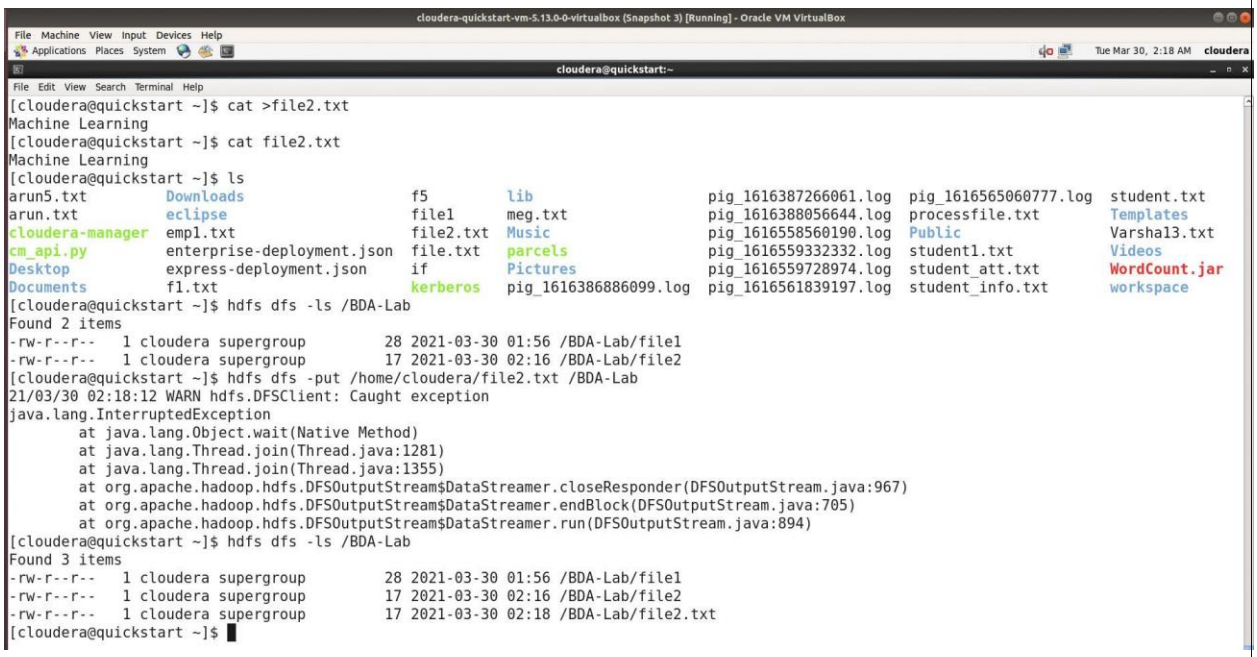
4 . c opy F romL oc al: T his command is used to copy file from local file sy stem (/ home/ cloudera) to H DFS. T o demonstrate this command, a file by name file1 is created in local file sy stem (/ home/ cloudera) using cat command and copied the same file1 to H DFS by using copy FromL ocal command.



5 . c opy T oL oc al: T his command is used to copy file from H DFS to local file sy stem (/ home/ cloudera). T o demonstrate this command, a file by name file1 is created in H DFS copied to local file sy stem (/ home/ cloudera). Note: Delete file1 in local file sy stem (/ home/ cloudera).

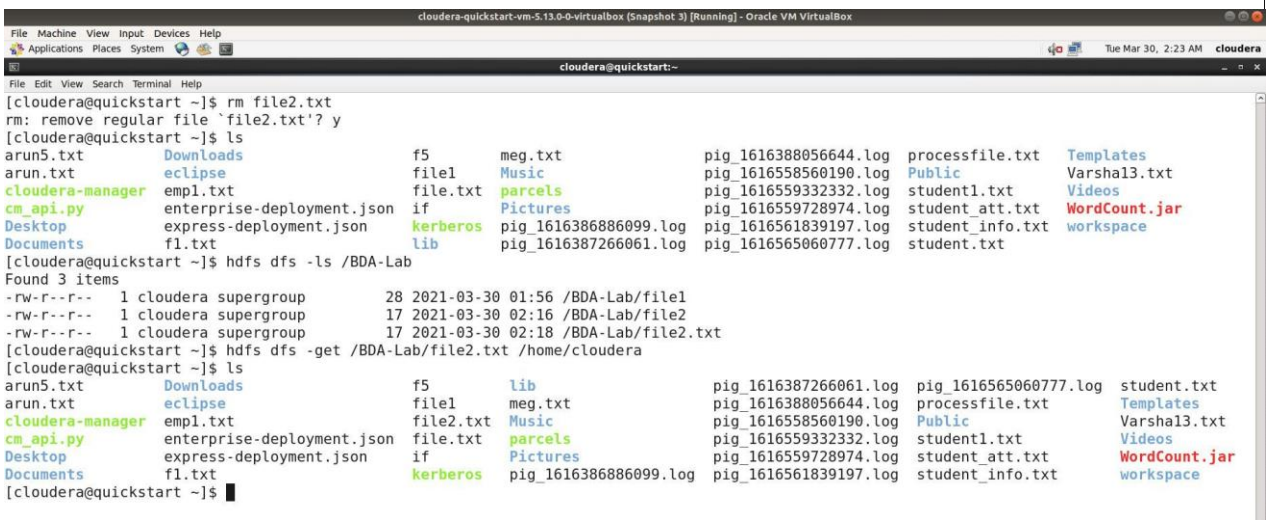


6 . put: T his command is used copy files from local file sy stem to H DFS



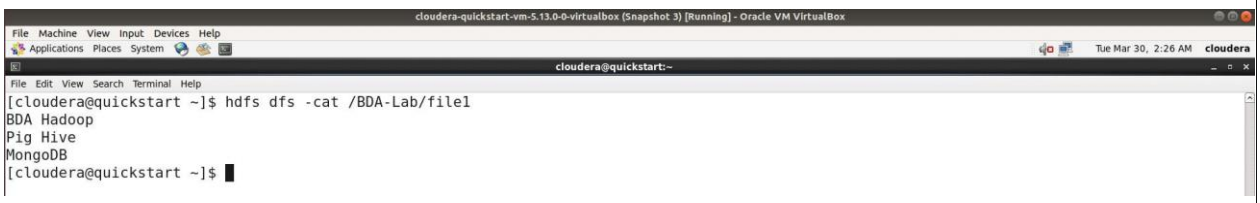
N ote: T he command copy FromL ocal is similar to put command, except that the sou rce is restricted to a local file reference.

7 . get: T his command is used copy files from H DFS to local file sy stem.

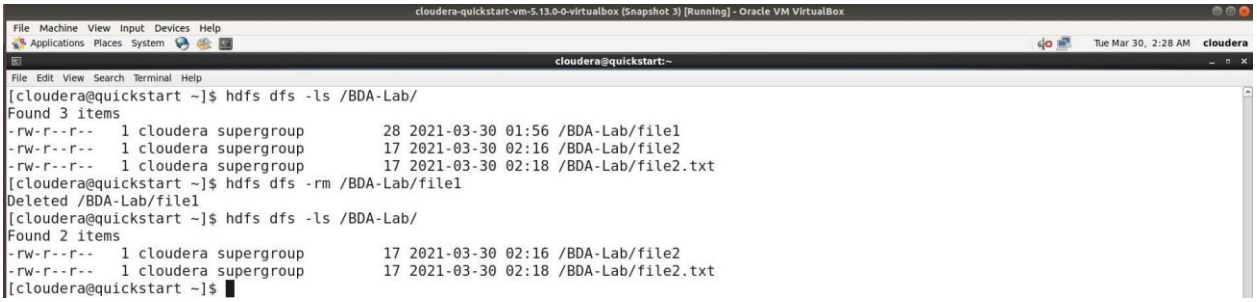


N ote: T he command copy T oL ocal is similar to get command, except that the dest ination is restricted to a local file reference.

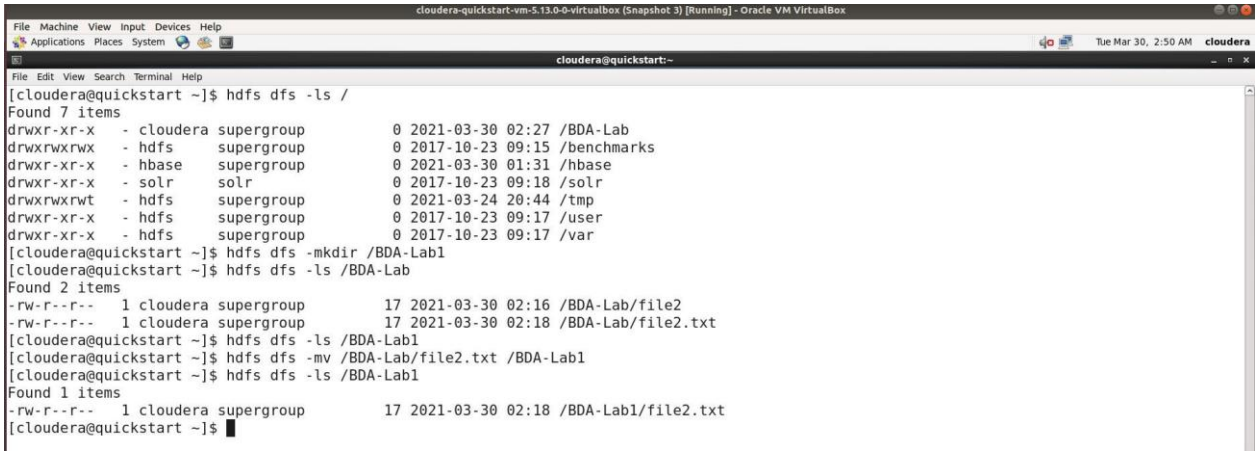
8. cat on HDFS: Retrieving files from HDFS



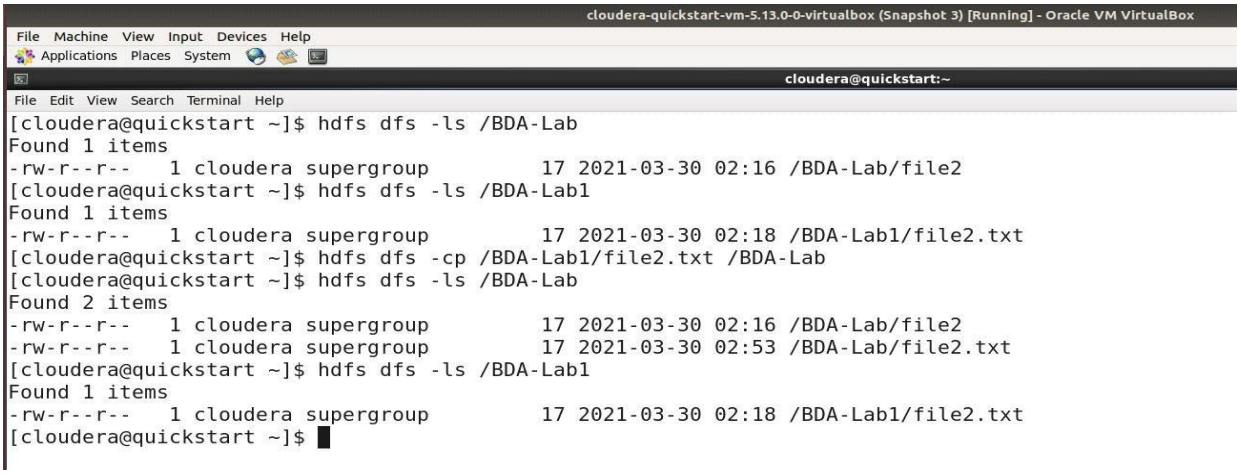
9. rm on HDFS: Deleting file from HDFS



10 . mv : T his command is used to move files across directories in H DFS



11. c p: T his command is used to copy files across directories in H DFS



Week-4: Run a basic Word Count Map-Reduce program to understand Map-Reduce Paradigm. Find the number of occurrence of each word appearing in the input file(s).

Step-1: Open Virtual box and then start **cloudera**

quickstart Step-2: Open **Eclipse** present on the cloudera

desktop Step-3: Creating Java Project

3.1 : File-> New -> Project -> **Java Project** -> Next

("WordCount" is the Project name)

Step-4: Adding the Hadoop libraries to the Project

4.1 : Click on **Add External Jars** button, the, File File System > usr > lib> **Hadoop**, select all the libraray (jar) files and then click OK button

4.2 : Now Click on Add External Jars button, the, File File System > usr > lib> Hadoop>**client**, select all the libraray (jar) files and then click OK button

4.3 : Click **finish** button

Step-5: Create Java MapperReduce program

5.1 : In the explorer panel Right click on "src" folder of the project WordCount New> Class> **Name** textfield give as "WordCount" and click Finish button.

5.2 : Type the code for WordCount program with import files, Mapper class, Reducer class Driver class with main method

Step-6: Export the project as JAR

6.1 : Right click WordCount project and select "Export" >> Java >> Jar file >> Next>> in the JAR file textfield give as /home/cloudera/WordCount.jar, click Finish button>> OK

6.2 : Open **cloudera@quickstart** terminal and verify the jar file using **ls** command Step-7: Create the input file for the MapReduce program by typing command

cat > /home/cloudera/f5

Verify the data contents by **cat /home/cloudera/f5**

Step-8: Move the input file created in local system to hdfs store by

hdfs dfs -put /home/cloudera/f5 /WordCount/

view the contents of the file moved to hdfs by typing command

hdfs dfs -cat /WordCount/f5

Step-9: Run MapReduce program on Hadoop by typing command

hadoop jar **/home/cloudera/WordCount.jar WordCount /WordCount/f5 /out24**

Each time you run the above command; you need to give different name for the output directory.

Step-10: View the output directory content by hdfs dfs -ls /out24 of the program/job executed,

hdfs dfs -cat /out24/part-r-00000

Example: WordCount

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

Source Code:

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


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

```
public class WordCount {


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


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


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


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


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

```
for (IntWritable val : values) {
```

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

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

```
[cloudera@quickstart ~]$ ls
arun5.txt                           eclipse                       file2.txt     parcels
arun.txt                            emp1.txt                      file.txt      Pictures
cloudera-manager                    enterprise-deployment.json    if            pig_1616386886099.log
cm_api.py                           express-deployment.json       kerberos      pig_1616387266061.log
CRND0103-2015-AK_Fairbanks_11_NE.txt f1                           lib           pig_1616388056644.log
Desktop                             f1.txt                        meg.txt       pig_1616558560190.log
Documents                           f5                            Music         pig_1616559332332.log
Downloads                           file1                         MyMaxMin.jar  pig_1616559728974.log
[cloudera@quickstart ~]$ cat file2.txt

[ ctoude ragquicksta rt -J S cat f1

!ciouderaqquicKstart -li cat f5


 cTouderagquicKstart -li cds

[cloudera@quickstart ~]$ clear
[cloudera@quickstart ~]$ cat f5


[cloudera@quickstart ~]$ hdfs dfs -mkdir /WordCount
[cloudera@quickstart ~]$ hdfs dfs -put /home/cloudera/f5 /WordCount
[cloudera@quickstart ~]$ hdfs dfs -cat /WordCount/f5
```

pig 61656 0b0777 log a a13.txt

Public WordCount.jar

st udent I .txt workspace

student att.txt

student_info.txt

student.txt

2I/ 04/05*2.5 4.2 01'WARN' napreduce 13 obResour ceUp\oader: RHadoopecDmman-dT ne0 ptTon pa rsEng not per fa rned . I ript ement the Tool interface and exec

```
File  Machine  View  Input  Devices  Help
Applications  Places  System
ute your application with ToolRunner to remedy this.
21/04/05 23:42:01 INFO input.FileInputFormat: Total input paths to process : 1
21/04/05 23:42:02 INFO mapreduce.JobSubmitter: number of splits:1
21/04/05 23:42:02 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1617681225311_0002
21/04/05 23:42:02 INFO impl.YarnClientImpl: Submitted application application_1617681225311_0002
21/04/05 23:42:02 INFO mapreduce.Job: The url to track the job: http://quickstart.cloudera:8088/proxy/application_1617681225311_0002/
21/04/05 23:42:02 INFO mapreduce.Job: Running job: job_1617681225311_0002
21/04/05 23:42:08 INFO mapreduce.Job: Job job_1617681225311_0002 running in uber mode : false
21/04/05 23:42:08 INFO mapreduce.Job:  map 0% reduce 0%
21/04/05 23:42:16 INFO mapreduce.Job:  map 100% reduce 0%
21/04/05 23:42:22 INFO mapreduce.Job:  map 100% reduce 100%
21/04/05 23:42:22 INFO mapreduce.Job: Job job_1617681225311_0002 completed successfully
21/04/05 23:42:22 INFO mapreduce.Job: Counters: 49
        File System Counters
                FILE: Number of bytes read=36
                FILE: Number of bytes written=286743
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=145
                HDFS: Number of bytes written=18
                HDFS: Number of read operations=6
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
        Job Counters
                Launched map tasks=1


                Total'time spent'by'all reduces in'occupied'slots'(msl=3532

                Total time spent by all reduce tasks (ms)=3532
                Total vcore-milliseconds taken by all map tasks=4483
```

```
                Total vcore-milliseconds taken by all reduce tasks=3532
                Total megabyte-milliseconds taken by all map tasks=4590592
                Total megabyte-milliseconds taken by all reduce tasks=3616768
        Map-Reduce Framework
                Map output records=9
                Map output bytes=72
```

Input spa it bytes=109

```
                Combine output records=3
                Reduce input groups=3
                Reduce shuffle bytes=36
                Reduce input records=3
                Reduce output records=3
                Spilled Records=6
                Shuffled Maps =1
                Failed Shuffles=0

                GC time elapsed (ms)=140
                CPU time spent (ms)=870
                Physical memory (bytes) snapshot=344375296
                Virtual memory (bytes) snapshot=3015176192
                Total committed heap usage (bytes)=226365440
```

Bytes Read=36

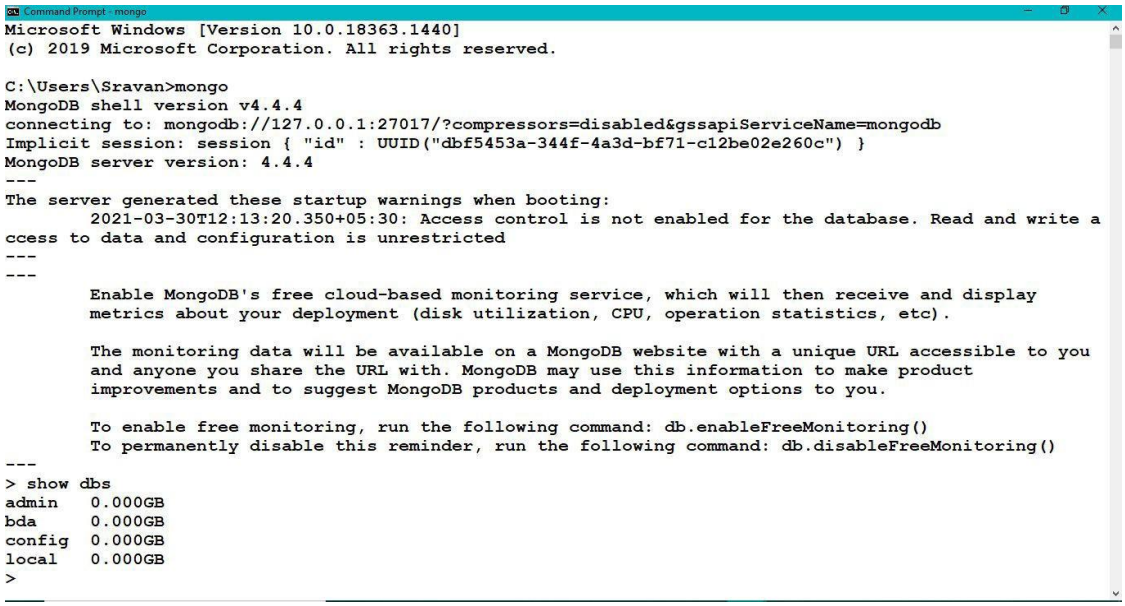
File Output Fo rmat Counte rs

[ctoude raequinks ta rt -1$ hdfs dT s -Is /out24

```
-rw-r--r--   1 cloudera supergroup          0 2021-04-05 23:42 /out24/_SUCCESS
-rw-r--r--   1 cloudera supergroup         18 2021-04-05 23:42 /out24/part-r-00000
[cloudera@quickstart ~]$ hdfs dfs -cat /out24/part-r-00000
```

[ctoude ragquinks ta rt - J G

Week-5: Perform I ncremental Map-Reduce using MongoD



To perform map- reduce operations, MongoDB provides the mapReduce command and, inthe mongo shell, the db.collection.mapReduce() wrapper method.

If the map- reduce data set is constantly growing, you may want to perform an incremental map- reduce rather than performing the map- reduce operation over the entire data set eachtime.

To perform incremental map- reduce:

1. Run a map- reduce job over the current collection and output the result to a separate collection.
2. When you have more data to process, run subseq uent map- reduce jobs with:
    - the q uery parameter that specifies conditions that match *on ly* the new documents.
    - the out parameter that specifies the reduce action to merge the new results into the existing output collection.

Consider the following example where you schedule a map- reduce operation on a usersessions collection to run at the end of each day.

D ata Setup

The usersessions collection contains documents that log users' sessions each day, for example:

```
Command Prompt - mongo
> use MONGODB
switched to db MONGODB
> db.usersessions.insertMany([
...     { userid: "a", start: ISODate('2020-03-03 14:17:00'), length: 95 },
...     { userid: "b", start: ISODate('2020-03-03 14:23:00'), length: 110 },
...     { userid: "c", start: ISODate('2020-03-03 15:02:00'), length: 120 },
...     { userid: "d", start: ISODate('2020-03-03 16:45:00'), length: 45 },
...     { userid: "a", start: ISODate('2020-03-04 11:05:00'), length: 105 },
...     { userid: "b", start: ISODate('2020-03-04 13:14:00'), length: 120 },
...     { userid: "c", start: ISODate('2020-03-04 17:00:00'), length: 130 },
...     { userid: "d", start: ISODate('2020-03-04 15:37:00'), length: 65 }
... ])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("606585fc630fa8f56c6a00c7"),
                ObjectId("606585fc630fa8f56c6a00c8"),
                ObjectId("606585fc630fa8f56c6a00c9"),
                ObjectId("606585fc630fa8f56c6a00ca"),
                ObjectId("606585fc630fa8f56c6a00cb"),
                ObjectId("606585fc630fa8f56c6a00cc"),
                ObjectId("606585fc630fa8f56c6a00cd"),
                ObjectId("606585fc630fa8f56c6a00ce")
        ]
}
>
```

```
Command Prompt - mongo
> db.usersessions.find()
{ "_id" : ObjectId("606585fc630fa8f56c6a00c7"), "userid" : "a", "start" : ISODate("2020-03-03T14:17:00Z
"), "length" : 95 }
{ "_id" : ObjectId("606585fc630fa8f56c6a00c8"), "userid" : "b", "start" : ISODate("2020-03-03T14:23:00Z
"), "length" : 110 }
{ "_id" : ObjectId("606585fc630fa8f56c6a00c9"), "userid" : "c", "start" : ISODate("2020-03-03T15:02:00Z
"), "length" : 120 }
{ "_id" : ObjectId("606585fc630fa8f56c6a00ca"), "userid" : "d", "start" : ISODate("2020-03-03T16:45:00Z
"), "length" : 45 }
{ "_id" : ObjectId("606585fc630fa8f56c6a00cb"), "userid" : "a", "start" : ISODate("2020-03-04T11:05:00Z
"), "length" : 105 }
{ "_id" : ObjectId("606585fc630fa8f56c6a00cc"), "userid" : "b", "start" : ISODate("2020-03-04T13:14:00Z
"), "length" : 120 }
{ "_id" : ObjectId("606585fc630fa8f56c6a00cd"), "userid" : "c", "start" : ISODate("2020-03-04T17:00:00Z
"), "length" : 130 }
{ "_id" : ObjectId("606585fc630fa8f56c6a00ce"), "userid" : "d", "start" : ISODate("2020-03-04T15:37:00Z
"), "length" : 65 }
```

I nitial Map-Reduce of Current Collection

Run the first map- reduce operation as follows:

1. Define the map function that maps the userid to an object that contains the fields total_ time, count, and avg_ time:

2. Define the corresponding reduce function with two arguments key and values to calculate the total time and the count. The key corresponds to the userid, and
   the values is an array whose elements corresponds to the individual objects mapped to the userid in the mapFunction

3. Define the finalize function with two arguments key and reducedValue. The function modifies the reducedValue document to add another field average and returns the modified document.

```
Command Prompt - mongo
> var mapFunction = function() {
...      var key = this.userid;
...      var value = { total_time: this.length, count: 1, avg_time: 0 };
...
...      emit( key, value );
... };
> var reduceFunction = function(key, values) {
...
...     var reducedObject = { total_time: 0, count:0, avg_time:0 };
...
...     values.forEach(function(value) {
...        reducedObject.total_time += value.total_time;
...        reducedObject.count += value.count;
...     });
...
...     return reducedObject;
... };
> var finalizeFunction = function(key, reducedValue) {
...
...     if (reducedValue.count > 0)
...        reducedValue.avg_time = reducedValue.total_time / reducedValue.count;
...
...     return reducedValue;
... };
>
```

4. Perform map- reduce on the usersessions collection using the mapFunction, the reduceFunction, and the finalizeFunction functions. Output the results to a collection session_ stats. If the session_ stats collection already exists, the operation will replace the contents:

5. Q uery the session_ stats collection to verify the results:

```
Command Prompt - mongo
> db.usersessions.mapReduce(
...     mapFunction,
...     reduceFunction,
...     {
...       out: "session_stats",
...       finalize: finalizeFunction
...     }
... )
{ "result" : "session_stats", "ok" : 1 }
> db.session_stats.find().sort( { _id: 1 } )
{ "_id" : "a", "value" : { "total_time" : 200, "count" : 2, "avg_time" : 100 } }
{ "_id" : "b", "value" : { "total_time" : 230, "count" : 2, "avg_time" : 115 } }
{ "_id" : "c", "value" : { "total_time" : 250, "count" : 2, "avg_time" : 125 } }
{ "_id" : "d", "value" : { "total_time" : 110, "count" : 2, "avg_time" : 55 } }
>
```

6 . Subsequent I ncremental Map-Reduce

L ater, as the usersessions collection grows, you can run additional map- reduce operations.For example, add new documents to the usersessions collection:

```
Select Command Prompt - mongo
> db.usersessions.insertMany([
...     { userid: "a", ts: ISODate('2020-03-05 14:17:00'), length: 130 },
...     { userid: "b", ts: ISODate('2020-03-05 14:23:00'), length: 40 },
...     { userid: "c", ts: ISODate('2020-03-05 15:02:00'), length: 110 },
...     { userid: "d", ts: ISODate('2020-03-05 16:45:00'), length: 100 }
... ])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("606589dd630fa8f56c6a00cf"),
                ObjectId("606589dd630fa8f56c6a00d0"),
                ObjectId("606589dd630fa8f56c6a00d1"),
                ObjectId("606589dd630fa8f56c6a00d2")
        ]
}
```

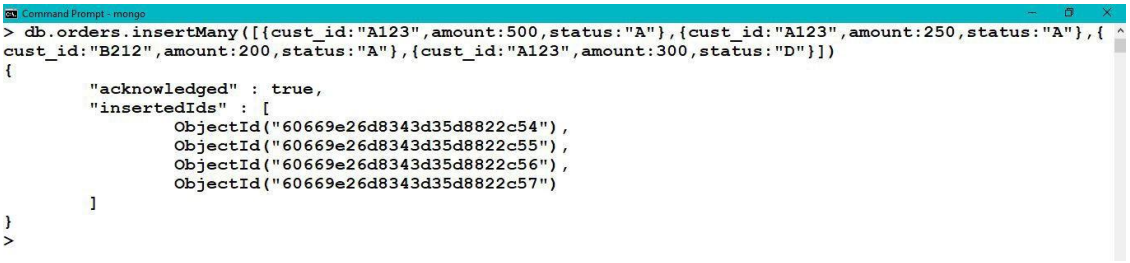
At the end of the day, perform incremental map- reduce on the usersessions collection, butuse the q uery field to select only the new documents. Output the results to the collection session_ stats, but reduce the contents with the results of the incremental map-reduce:

```
Select Command Prompt - mongo
> db.usersessions.find()
{ "_id" : ObjectId("606585fc630fa8f56c6a00c7"), "userid" : "a", "start" : ISODate("2020-03-03T14:17:00Z
"), "length" : 95 }
{ "_id" : ObjectId("606585fc630fa8f56c6a00c8"), "userid" : "b", "start" : ISODate("2020-03-03T14:23:00Z
"), "length" : 110 }
{ "_id" : ObjectId("606585fc630fa8f56c6a00c9"), "userid" : "c", "start" : ISODate("2020-03-03T15:02:00Z
"), "length" : 120 }
{ "_id" : ObjectId("606585fc630fa8f56c6a00ca"), "userid" : "d", "start" : ISODate("2020-03-03T16:45:00Z
"), "length" : 45 }
{ "_id" : ObjectId("606585fc630fa8f56c6a00cb"), "userid" : "a", "start" : ISODate("2020-03-04T11:05:00Z
"), "length" : 105 }
{ "_id" : ObjectId("606585fc630fa8f56c6a00cc"), "userid" : "b", "start" : ISODate("2020-03-04T13:14:00Z
"), "length" : 120 }
{ "_id" : ObjectId("606585fc630fa8f56c6a00cd"), "userid" : "c", "start" : ISODate("2020-03-04T17:00:00Z
"), "length" : 130 }
{ "_id" : ObjectId("606585fc630fa8f56c6a00ce"), "userid" : "d", "start" : ISODate("2020-03-04T15:37:00Z
"), "length" : 65 }
{ "_id" : ObjectId("606589dd630fa8f56c6a00cf"), "userid" : "a", "ts" : ISODate("2020-03-05T14:17:00Z"),
 "length" : 130 }
{ "_id" : ObjectId("606589dd630fa8f56c6a00d0"), "userid" : "b", "ts" : ISODate("2020-03-05T14:23:00Z"),
 "length" : 40 }
{ "_id" : ObjectId("606589dd630fa8f56c6a00d1"), "userid" : "c", "ts" : ISODate("2020-03-05T15:02:00Z"),
 "length" : 110 }
{ "_id" : ObjectId("606589dd630fa8f56c6a00d2"), "userid" : "d", "ts" : ISODate("2020-03-05T16:45:00Z"),
 "length" : 100 }
>
```

Q uery the session_ stats collection to verify the results:

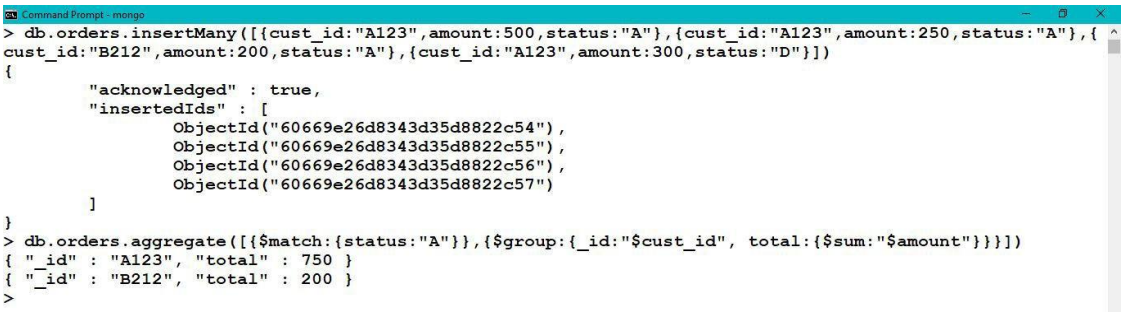
```
Command Prompt - mongo
> db.usersessions.mapReduce(
...     mapFunction,
...     reduceFunction,
...     {
...       query: { ts: { $gte: ISODate('2020-03-05 00:00:00') } },
...       out: { reduce: "session_stats" },
...       finalize: finalizeFunction
...     }
... );
{ "result" : "session_stats", "ok" : 1 }
> db.session_stats.find().sort( { _id: 1 } )
{ "_id" : "a", "value" : { "total_time" : 330, "count" : 3, "avg_time" : 110 } }
{ "_id" : "b", "value" : { "total_time" : 270, "count" : 3, "avg_time" : 90 } }
{ "_id" : "c", "value" : { "total_time" : 360, "count" : 3, "avg_time" : 120 } }
{ "_id" : "d", "value" : { "total_time" : 210, "count" : 3, "avg_time" : 70 } }
>
```

Week-6 : Write and Execute simple queries to perform Aggregation and I ndexing



Aggregation

Aggregation operation process data records and return computed results. Aggregation operations group values from multiple documents together and can perform a variety of operations on the grouped data to return a single result.



For example:

First Stage: The $ match stage filters the documents by the status field and passes to the next stage those documents that have status eq ual to “A”.

Second Stage: The $ group stage groups the documents by the cust_ id field to calculate the sum of the amount for each uniq ue cust_ id

I ndexing

Indexes provide users with an efficient way of q uerying data. When q uerying data without indexes, the q uery will have to search for all the records within a database to find data that match the q uery.
In MongoDB, q uerying without indexes is called a collection scan. A collection scan will:

- Result in various performance bottlenecks
- Significantly slow down your application

Fortunately, using indexes fixes both these issues. By limiting the number of documents to be q ueried, you' ll increases the overall performance of the application.

What are indexes in MongoD B ?
Indexes are special data structures that store a small part of the Collection' s data in a way that can be q ueried easily.

In simplest terms, indexes store the values of the indexed fields outside the table or collection and keep track of their location in the disk. These values are used to order the indexed fields. This ordering helps to perform eq uality matches and range- based q uery operations efficiently. In MongoDB, indexes are defined in the collection level and indexes on any field or subfield of the documents in a collection are supported.

Creating indexes

When creating documents in a collection, MongoDB creates a uniq ue index using the _ id field. MongoDB refers to this as the **D efault _ id I ndex**. This default index cannot be dropped from the collection.
When q uerying the test data set, you can see the _ id field which will be utilized as the default index. Now let' s create an index. To do that, you can use the **createI ndex** method.

When creating an index, you need to define the field to be indexed and the direction of the key (1 or 1) to indicate ascending or descending order.

Another thing to keep in mind is the index names. By default, MongoDB will generate index names by concatenating the indexed keys with the direction of each key in the index using an underscore as the separator. For example: {name: 1} will be created as name_ 1.

The best option is to use the name option to define a custom index name when creating an index. Indexes cannot be renamed after creation. (The only way to rename an index is to first drop that index, which we show below, and recreate it using the desired name.) L et' s create an index using the name field in the studentgrades collection and name it as **student name index**.

Finding indexes

Y ou can find all the available indexes in a MongoDB collection by using the **getI ndexes** method. This will return all the indexes in a specific collection.

L et' s view all the indexes in the studentgrades collection using the following command:

The output contains the **default _ id index** and the user- created index **student name index**

D ropping indexes

To delete an index from a collection, use the **dropI ndex** method while specifying the index name to be dropped.

L et' s remove the user- created index with the index name **student name index**, as shown below.

Y ou can also use the index field value for removing an index without a defined

```
db.studentgrades.dropIndex({name:1})
```

The **dropI ndexes** command can also drop all the indexes excluding the default _ id index.

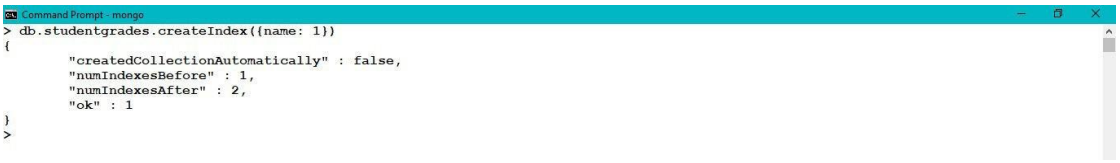
Common MongoD B index ty pes

MongoDB provides different types of indexes that can be utilized according to user needs.
Here are the most common ones:

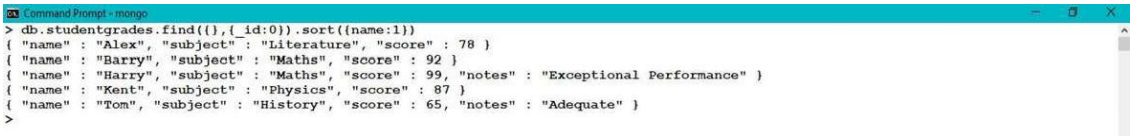
- Single field index
- Compound index
- Multikey index

Single Field I ndex

These user- defined indexes use a single field in a document to create an index in an ascending ordescending sort order (1 or - 1). In a single field index, the sort order of the index key does not have animpact because MongoDB can traverse the index in either direction.



The above index will sort the data in ascending order using the name field. Y ou can use the **sort()** method to see how the data will be represented in the index.



Compound Field I ndex

Y ou can use multiple fields in a MongoDB document to create a compound index. This type of indexwill use the first field for the initial sort and then sort by the preceding fields.

In the above compound index, MongoDB will:

- First sort by the subject field
- Then, within each subject value, sort by grade

The index would create a data structure similar to the

```
Command Prompt - mongo
> db.studentgrades.find({},{_id:0}).sort({subject:1, score:-1})
{ "name" : "Tom", "subject" : "History", "score" : 65, "notes" : "Adequate" }
{ "name" : "Alex", "subject" : "Literature", "score" : 78 }
{ "name" : "Harry", "subject" : "Maths", "score" : 99, "notes" : "Exceptional Performance" }
{ "name" : "Barry", "subject" : "Maths", "score" : 92 }
{ "name" : "Kent", "subject" : "Physics", "score" : 87 }
>
```

Multikey I ndex

MongoDB supports indexing array fields. When you create an index for a field containing an array, MongoDB will create separate index entries for every element in the array. These multikey indexes enable users to q uery documents using the elements within the array.

MongoDB will automatically create a multikey index when encountered with an array field without req uiring the user to explicitly define the multikey type.

L et' s create a new data set containing an array field to demonstrate the creation of a multikey index.

```
Command Prompt - mongo
> db.createCollection("studentperformance")
{ "ok" : 1 }
> db.studentperformance.insertMany(
... [
... {name: "Barry", school: "ABC Academy", grades: [85, 75, 90, 99] },
... {name: "Kent", school: "FX High School", grades: [74, 66, 45, 67]},
... {name: "Alex", school: "XYZ High", grades: [80, 78, 71, 89]},
... ]
... )
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("606ac37bc94f3a21316aadd8"),
                ObjectId("606ac37bc94f3a21316aadd9"),
                ObjectId("606ac37bc94f3a21316aadda")
        ]
}
> db.studentperformance.find()
{ "_id" : ObjectId("606ac37bc94f3a21316aadd8"), "name" : "Barry", "school" : "ABC Academy", "grades" : [ 85, 75, 90, 99 ] }
{ "_id" : ObjectId("606ac37bc94f3a21316aadd9"), "name" : "Kent", "school" : "FX High School", "grades" : [ 74, 66, 45, 67 ] }
{ "_id" : ObjectId("606ac37bc94f3a21316aadda"), "name" : "Alex", "school" : "XYZ High", "grades" : [ 80, 78, 71, 89 ] }
>
```

Now let' s create an index using the grades field.

```
Command Prompt - mongo
> db.studentperformance.createIndex({grades:1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

The above code will automatically create a Multikey index in MongoDB. When you q uery for a document using the array field (grades), MongoDB will search for the first element of the array defined in the **find()** method and then search for the whole matching q uery.

For instance, let' s consider the following find q uery:

```
Command Prompt - mongo
> db.studentperformance.find({grades: [80, 78, 71, 89]}, {_id: 0})
{ "name" : "Alex", "school" : "XYZ High", "grades" : [ 80, 78, 71, 89 ] }
>
```

Initially, MongoDB will use the multikey index for searching documents where the grades array contains the first element (80) in any position. Then, within those selected documents, the documents with all the matching elements will be selected.

Week-7	Pig commands – Pig Latin - Relational Operators: Loading, Storing, Diagnostic Operators (Dump, Describe, Illustrate & Explain)

7a For the given Student dataset and Employee dataset, perform Relational operations like Loading, Storing, Diagnostic Operations (Dump, Describe, Illustrate & Explain) in Hadoop Pig framework using Cloudera

Student ID	First Name	Age	City	CGPA
001	Jagruthi	21	Hyderabad	9.1
002	Praneeth	22	Chennai	8.6
003	Sujith	22	Mumbai	7.8
004	Sreeja	21	Bengaluru	9.2
005	Mahesh	24	Hyderabad	8.8
006	Rohit	22	Chennai	7.8
007	Sindhu	23	Mumbai	8.3

Employee ID	Name	Age	City
001	Angelina	22	LosAngeles
002	Jackie	23	Beijing
003	Deepika	22	Mumbai
004	Pawan	24	Hyderabad
005	Rajani	21	Chennai
006	Amitabh	22	Mumbai

Step-1: **Create a Directory** in HDFS with the name **pigdir** in the required path using **mkdir**:

$ hdfs dfs -mkdir /bdalab/pigdir

Step-2: The input file of Pig contains each tuple/record in individual lines with the entities separated by a delimiter (“,”).

In the local file system, **create an input file student_data.txt** containing data as shown below.	In the local file system, **create an input file employee_data.txt** containing data as shown below.
001,Jagruthi,21,Hyderabad,9.1 002,Praneeth,22,Chennai,8.6 003,Sujith,22,Mumbai,7.8 004,Sreeja,21,Bengaluru,9.2 005,Mahesh,24,Hyderabad,8.8 006,Rohit,22,Chennai,7.8 007,Sindhu,23,Mumbai,8.3	001,Angelina,22,LosAngeles 002,Jackie,23,Beijing 003,Deepika,22,Mumbai 004,Pawan,24,Hyderabad 005,Rajani,21,Chennai 006,Amitabh,22,Mumbai

Step-3: **Move the file** from the local file system to HDFS using **put (Or) copyFromLocal** command and verify using -cat command

$ hdfs dfs -put /home/cloudera/pigdir/student_data /bdalab/pigdir/

$ hdfs dfs -cat / bdalab/pigdir/student_data

$ hdfs dfs -put /home/cloudera/pigdir/employee_data /bdalab/pigdir/

$ hdfs dfs -cat /bdalab/pigdir/employee_data

Step-4: **Apply Relational Operator – LOAD to load the data** from the file student_data.txt into Pig byexecuting the following Pig Latin statementin the **Gruntshell**. Relational Operators are **NOT case sensitive.**

$ pig => will direct to **grunt> shell**

grunt> student = LOAD' /bdalab/pigdir/student_data.txt' USING PigStorage(',') as (id:int, name:chararray, age:int, city:chararray, cgpa:double);

grunt> employee = LOAD' /bdalab/pigdir/employee_data.txt' USING PigStorage(',') as (id:int, name:chararray, age:int, city:chararray);

Step-5: **Apply Relational Operator – STORE** to **Store the relation** in the HDFS directory "/pig_output/" as shown below.

grunt> STOREstudent INTO' /bdalab/pigdir/pig_output/ 'USING PigStorage (',');

grunt> STORE employee INTO ' /bdalab/pigdir/pig_output/ ' USING PigStorage (',');

Step-6: **Verify the stored data** as shown below

$ hdfs dfs -ls /bdalab/pigdir/pig_output/

$ hdfs dfs -cat /bdalab/pigdir/pig_output/**part-m-00000**

Step-7: **Apply Relational Operator – Diagnostic Operator – DUMP to Print the contents of the relation.**

grunt> Dump student

grunt> Dump employee

Step-8: **Apply Relational Operator – Diagnostic Operator – DESCIBE to View theschema of a relation.**

grunt> Describe student

grunt> Describe employee

Step-9: **Apply Relational Operator – Diagnostic Operator – EXPLAIN to Display the logical, physical, and MapReduce execution plans** of a relation using **Explain** operator

grunt> Explain student

grunt> Explain employee

Step-9: **Apply Relational Operator – Diagnostic Operator – ILLUSTRATE to give the step-by-step execution of a sequence of statements**

grunt> Illustrate student

grunt> Illustrate employee

7b	For the given Student dataset and Employee dataset, perform Relational operator – GROUP operations in Hadoop Pig framework using Cloudera

The **GROUP** operator is used to group the data in one or more relations. It collects the data having the same key.

grunt> Group_data = GROUP Relation_name BY Key;

Step-1: **Groupthe records/tuples** inthe relation byageusing **GROUP** command andverify.

grunt> group_std = GROUP student BY age;

grunt> Dump group_std;

grunt> group_emp = GROUP employee BY city;

grunt> Dump group_emp;

Step-2: View **Schema of the table after grouping** the data using the describe command as shown below.

grunt> Describe group_std;

group_std: {group: int,student: {(id:int, name:chararray, age:int, city:chararray, cgpa:float)}}

grunt> Describe group_emp;

group_emp: {group: int,employee: {(id: int,name: chararray,age:int,city: chararray)}}

Step-3: **Group by multiple columns** of the relation by age and city and verify the content.

grunt> groupmultiple_std = GROUP student BY (age, city);

grunt> Dump groupmultiple_std

grunt> groupmultiple_emp = GROUP employee BY (age, city);

grunt> Dump groupmultiple_emp

Step-4: **Group by All columns** of the relation and verifythe content.

grunt> groupall_std = GROUP student All;

grunt> Dump groupall_std

grunt> groupall_emp = GROUP employee All;

grunt> Dump groupall_emp

Step-5: **Combinedly Group the records/tuples** of the relations student_data and employee_data with the key age and then verify the result.

grunt> cogroup_stdemp = COGROUP student_data by age, employee_data by age;

grunt> Dump cogroup_stdemp

Week-7	Pig commands – Pig Latin - Relational Operator: Join
7c	For the given Student data and Student attendance datasets, perform Relational operations like JOIN (Self-join, Inner-join, Outer-join: Left join, Right join, and Full join) in Hadoop Pig framework using Cloudera.

Student_data				
Student ID	**Name**	**Age**	**City**	**CGPA**
1	Jagruthi	21	Hyderabad	9.1
2	Praneeth	22	Chennai	8.6
3	Sujith	22	Mumbai	7.8
4	Sreeja	21	Bengaluru	9.2
5	Mahesh	24	Hyderabad	8.8
6	Rohit	22	Chennai	7.8
7	Sindhu	23	Mumbai	8.3

Student_att			
Student ID	**Name**	**Action**	**Time**
1	Jagruthi	Joined	9:10:10
4	Sreeja	Joined	9:10:24
6	Rohit	Joined	9:11:15
7	Sindhu	Joined	9:12:25
8	Sai	Joined	9:14:18
9	Meghana	Joined	9:15:25

Step-1: **Create a Directory** in HDFS with the name **pigdir** in the required path using **mkdir**:

$ hdfs dfs -mkdir /bdalab/pigdir

Step-2: The input file of Pig contains each tuple/record in individual lines with the entities separated by a delimiter (“,”).

In the local file system, **create an input file student_data.txt** containing data as shown below.	In the local file system, **create an input file student_att.txt** containing data as shown below.
1,Jagruthi,21,Hyderabad,9.1 2,Praneeth,22,Chennai,8.6 3,Sujith,22,Mumbai,7.8 4,Sreeja,21,Bengaluru,9.2 5,Mahesh,24,Hyderabad,8.8 6,Rohit,22,Chennai,7.8 7,Sindhu,23,Mumbai,8.3	1,Jagruthi,joined,9:10:10 4,Sreeja,joined,9:10:24 6,Rohit,joined,9:11:15 7,Sindhu,joined,9:12:25 8,Sai,joined,9:14:18 9,Meghana,joined,9:15:25

Step-3: **Move the file** from the local file system to HDFS using **put (Or) copyFromLocal** command and verify using -cat command

$ hdfs dfs -put /home/cloudera/pigdir/student_data /bdalab/pigdir/

$ hdfs dfs -cat / bdalab/pigdir/student_data

$ hdfs dfs -put /home/cloudera/pigdir/student_att /bdalab/pigdir/

$ hdfs dfs -cat /bdalab/pigdir/student_att

Step-4: Open Pig in **Grunt shell** and execute the following Pig Latin statement.

Apply Relational Operator – LOAD to load the data into Relation name std_data from the file student_data.txt into Relational Operators are **NOT case sensitive.**

$ pig => will direct to **grunt>**

grunt> std_data =LOAD' /bdalab/pigdir/student_data' USING PigStorage(',') as (id:int, name:chararray, age:int, city:chararray, cgpa:double);

grunt> DUMP std_data;

(1,Jagruthi,21,Hyderabad,9.1)
(2,Praneeth,22,Chennai,8.6)
(3,Sujith,22,Mumbai,7.8)
(4,Sreeja,21,Bengaluru,9.2)
(5,Mahesh,24,Hyderabad,8.8)
(6,Rohit,22,Chennai,7.8)
(7,Sindhu,23,Mumbai,8.3)

grunt> std_att = LOAD ' /bdalab/pigdir/student_att' USING PigStorage(',') as (id:int, name:chararray, action:chararray, time:chararray);

grunt> DUMP std_att;

(1,Jagruthi,joined,9:10:10)
(4,Sreeja,joined,9:10:24)
(6,Rohit,joined,9:11:15)
(7,Sindhu,joined,9:12:25)
(8,Sai,joined,9.14:18)
(9,Meghana,joined,9.15:25)

The **JOIN** operator is used to combine records from two or more relations. While performing a join operation, we declare one (or a group of) tuple(s) from each relation, as keys. When these keys match, the two tuples are matched, else the records are dropped. Joins can be of the following types –

- SELF-Join
- INNER-Join
- OUTER-Join – LEFT Join, RIGHT Join, and FULL Join

Step-5: **SELF-JOIN**, we will load the same data multiple times, under different aliases (names).

grunt> std1 = LOAD ' /bdalab/pigdir/student_data ' USING PigStorage(',') as (id:int, name:chararray, age:int, city:chararray, cgpa:float);

grunt> std2 = LOAD ' /bdalab/pigdir/student_data 'USING PigStorage(',') as (id:int, name:chararray, age:int, city:chararray, cgpa:float);

grunt> selfjoin_std_data = JOIN students1 BY id, students2 BY id;

grunt> dump selfjoin_std_data;

(1,Jagruthi,21,Hyderabad,9.1,1,Jagruthi,21,Hyderabad,9.1)

(2,Praneeth,22,Chennai,8.6,2,Praneeth,22,Chennai,8.6)
(3,Sujith,22,Mumbai,7.8,3,Sujith,22,Mumbai,7.8)
(4,Sreeja,21,Bengaluru,9.2,4,Sreeja,21,Bengaluru,9.2)
(5,Mahesh,24,Hyderabad,8.8,5,Mahesh,24,Hyderabad,8.8)
(6,Rohit,22,Chennai,7.8,6,Rohit,22,Chennai,7.8)
(7,Sindhu,23,Mumbai,8.3,7,Sindhu,23,Mumbai,8.3)

Step-6: **INNER JOIN - EQUI JOIN** creates a new relation by combining column values of two relations based upon the join-predicate. It returns rows when there is a match in both tables.

grunt> innerjoin_data_att = JOIN std_data BY id, std_att BY id;

grunt> dump std_data_att;

(1,Jagruthi,21,Hyderabad,9.1,1,Jagruthi,joined,9:10:10)
(4,Sreeja,21,Bengaluru,9.2,4,Sreeja,joined,9:10:24)
(6,Rohit,22,Chennai,7.8,6,Rohit,joined,9:11:15)
(7,Sindhu,23,Mumbai,8.3,7,Sindhu,joined,9:12:25)

OUTER JOIN returns all the rows from at least one of the relations. An outer join operation is carried out in three ways –

- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN

Step-7: **LEFT OUTER JOIN** operation returns all rows from the left table, even if there are no matches in the right relation.

Note: Student_data is LEFT

grunt> outerleft_data_att = JOIN std_data BY id LEFT, std_att BY id;

grunt> DUMP outerleft_data_att

(1,Jagruthi,21,Hyderabad,9.1,1,Jagruthi,joined,9:10:10)
(2,Praneeth,22,Chennai,8.6,,,,)
(3,Sujith,22,Mumbai,7.8,,,,)
(4,Sreeja,21,Bengaluru,9.2,4,Sreeja,joined,9:10:24)
(5,Mahesh,24,Hyderabad,8.8,,,,)
(6,Rohit,22,Chennai,7.8,6,Rohit,joined,9:11:15)
(7,Sindhu,23,Mumbai,8.3,7,Sindhu,joined,9:12:25)

Note: Student_att is LEFT

grunt> outerleft_att_data = JOIN std_att BY id LEFT, std_data BY id;

grunt> DUMP outerleft_att_data;

(1,Jagruthi,joined,9:10:10,1,Jagruthi,21,Hyderabad,9.1)
(4,Sreeja,joined,9:10:24,4,Sreeja,21,Bengaluru,9.2)
(6,Rohit,joined,9:11:15,6,Rohit,22,Chennai,7.8)
(7,Sindhu,joined,9:12:25,7,Sindhu,23,Mumbai,8.3)

(8,Sai,joined,9.14:18,,,,,)
(9,Meghana,joined,9.15:25,,,,,)

Step-8: **RIGHT OUTER JOIN** operation returns all rows from the right table, even if there are no matches in the left table.

```
grunt> outerright_data_att = JOIN std_data BY id RIGHT, std_att BY id;

grunt> DUMP outerright_data_att;
```

(1,Jagruthi,21,Hyderabad,9.1,1,Jagruthi,joined,9:10:10)
(4,Sreeja,21,Bengaluru,9.2,4,Sreeja,joined,9:10:24)
(6,Rohit,22,Chennai,7.8,6,Rohit,joined,9:11:15)
(7,Sindhu,23,Mumbai,8.3,7,Sindhu,joined,9:12:25)
(,,,,,8,Sai,joined,9.14:18)
(,,,,,9,Meghana,joined,9.15:25)

Step-9: **FULL OUTER JOIN** operation returns rows when there is a match in one of the relations.

```
grunt> outerfull_data_att = JOIN std_data BY id FULL, std_att BY id;

grunt> DUMP outerfull_data_att;
```

(1,Jagruthi,21,Hyderabad,9.1,1,Jagruthi,joined,9:10:10)
(2,Praneeth,22,Chennai,8.6,,,,)
(3,Sujith,22,Mumbai,7.8,,,,)
(4,Sreeja,21,Bengaluru,9.2,4,Sreeja,joined,9:10:24)
(5,Mahesh,24,Hyderabad,8.8,,,,)
(6,Rohit,22,Chennai,7.8,6,Rohit,joined,9:11:15)
(7,Sindhu,23,Mumbai,8.3,7,Sindhu,joined,9:12:25)
(,,,,,8,Sai,joined,9.14:18)
(,,,,,9,Meghana,joined,9.15:25)

Week-8	Hive Data Definition Language (DDL) Commands
8a	Implement Data Definition Language (DDL) Commands for databases in Hadoop Hive framework using Cloudera.

- Open Virtual box and then start **cloudera quickstart Terminal** and type "hive" to launch hive shell

DDL Commands forDatabases

1) **CREATE** database Statementisused to create adatabasein Hive. Adatabase in Hive isa namespace or a collection or catalog of tables.
   Syntax: **CREATE DATABASE|SCHEMA [IF NOT EXISTS] database_name**
   [COMMENT database_comment]
   [LOCATION hdfs_path]
   [WITH DBPROPERTIES (property_name=property_value, ...)];

   [ ] are optional clauses. We can use SCHEMA in place of DATABASE in this command. The following query is executed to create a database named employee. If everything went good, you will see a 'OK' message, else you will see relevant error message.
   Simple creation
   hive> CREATE DATABASE facultycse;
   OK
   Time taken: 0.033 seconds
   hive> CREATE DATABASEfacultyece;

   Full creation
   hive> CREATE DATABASE IF NOT EXISTS employee COMMENT 'this is employee database' LOCATION '/user/hive/warehouse/hivedir/' WITH DBPROPERTIES ('creator'='Bhanu', 'date'='2020-12-07');

2) **SHOW** databases statement lists all the databases present in the metastore.
   Syn: **SHOW (DATABASES/SCHEMAS) [LIKE 'wildcards'];**
   - Wildcards in the regular expression can only be '*' for any character(s) or '|' for a choice. Examples are 'employees', 'emp*', 'emp*|*ees', all of which will match the database named 'employees':

hive> SHOW DATABASES; default employee facultycse facultyece	hive> SHOW DATABASES LIKE '*ee'; employee hive> SHOW DATABASES LIKE 'fac*'; facultycse facultyece

3) **DESCRIBE** database statement in Hiveshowsthename of Databasein Hive, itscomment(if set), its location, its owner name, owner type and its properties.
   Syn: **DESCRIBE DATABASE/SCHEMA [EXTENDED] db_name;**
   - EXTENDED can be used to get the database properties.

hive>DESCRIBE DATABASE facultycse;
facultycse hdfs://quickstart.cloudera:8020/user/hive/warehouse/faculty.db cloudera USER

hive>DESCRIBE DATABASE EXTENDED employee;
employee this is employee database hdfs://quickstart.cloudera:8020/user/hive/warehouse/ cloudera USER {date=2020-12-07, creator=Bhanu};

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

Syn: **USE db_name;**

hive> USE employee;

OK

5) **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.

Syn: **DROP (DATABASE|SCHEMA) [IF EXISTS] db_name [RESTRICT|CASCADE];**

hive> DROP DATABASE facultyece;

OK

hive> DROP DATABASE IF EXISTS facultycse CASCADE;

OK

6) **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) db_name SET DBPROPERTIES (property_name=property_value, ...);

hive> ALTER DATABASE employee SET DBPROPERTIES ('creator'='Bhanu Prasad', 'date'='07-12-2020');

employee this is employee database hdfs://quickstart.cloudera:8020 /user/hive/warehouse/hivedir/ cloudera USER {date= **07-12-2020**, creator=**Bhanu Prasad**};

Syn for changing Database owner:

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

hive> ALTER DATABASE employee SET OWNER **USER client**;
employee this is employee database hdfs://quickstart.cloudera:8020 /user/hive/warehouse/hivedir/ **client USER** {date= 07-12-2020, creator=Bhanu Prasad};

hive> ALTER DATABASE employee SET OWNER **ROLE Admin**;

employee this is employee database hdfs://quickstart.cloudera:8020 /user/hive/warehouse/hivedir/ **Admin ROLE** {date= 07-12-2020, creator=Bhanu Prasad};

7b	Implement Data Definition Language (DDL) Commands for tables in Hadoop Hive framework using Cloudera.

DDL Commands forTables

1) **CREATETABLE** statement in Hive isused to create atable with the givenname. If atable or view already exists with the same name, then the error is thrown. We can use IF NOT EXISTS to skip the error.

   Syn: **CREATETABLE[IFNOTEXISTS][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];

   hive> CREATE TABLE IF NOT EXISTS employee.emptable (emp_id STRING COMMENT 'This is Employee ID', emp_name STRING COMMENT 'This is Employee Name', emp_sal FLOAT COMMENT 'This is Employee Salary')
   COMMENT 'This table contains Employees Data'
   ROW FORMAT DELIMITED
   FIELDS TERMINATED BY ','
   STORED AS TEXTFILE;

2) **SHOW** tables statement in Hive lists all the base tables and views in the current database.

   Syn: **SHOW TABLES [IN database_name];**

   hive> SHOWTABLESINemployee;
   OK
   emptable

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

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

   hive> DESCRIBE employee.emptable;
   emp_id string This is Employee ID
   emp_name string This is Employee Name
   emp_sal float This is Employee Salary
   hive> DESCRIBE EXTENDED employee.emptable;
   hive> DESCRIBE FORMATTED employee.emptable;

4) **ALTER** table statement in Hive enables you to change the structure of an existing table, rename the table, add columns to the table, change the table properties, etc.

   Syntax for Rename a table:

ALTER TABLE table_name RENAME TO new_table_name;

hive> ALTER TABLE employee.emptable RENAME TO employee.facultytable;

Syn to Add columns to a table:

ALTER TABLE table_name ADD COLUMNS (column1, column2) ;

hive> ALTER TABLE employee.facultytable ADD COLUMNS (emp_post string COMMENT 'This is employee post', emp_age INT COMMENT 'This is employee age');

Syn to set table properties:

ALTER TABLE table_name SET TBLPROPERTIES ('property_key'='property_new_value');

hive> ALTER TABLE employee.facultytable SET TBLPROPERTIES ('table for'='faculty data');

5) **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.

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

   hive> DROP TABLE IF EXISTS employee.emptable PURGE;

   OK

6) **TRUNCATE** table statement in Hive removes all the rows from the table or partition.

   Syn: **TRUNCATE TABLE table_name;**

   hive> TRUNCATE TABLE employee.emptable;

   OK

Week-8	Hive Data Manipulation Language (DML) Commands
8	Implement Data Manipulation Language (DML) Commands for tables in Hadoop Hive framework using Cloudera.

- Open Virtual box and then start **cloudera quickstart Terminal** and type "hive" to launch hive shell

DML Commands for Tables

1) **LOAD** statement in Hive is used to copy/move data files into the locations corresponding to Hive tables.

Syn: **LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...)];**

LOCAL keyword = file path in the local file system.

LOCAL not specified = file path in the hdfs

OVERWRITE contents of the target table (or partition) will be deleted and replaced by the files otherwise contents are added to the table

hive> LOAD DATA LOCAL INPATH '/home/cloudera/HiveDir/emptextdata' INTO TABLE employee.facultytable;

OK

emptextdata contents

1,bob,25000.00,asstprof,35,male

2,mary,35000.00,assocprof,38,female

3,mike,50000.00,prof,45,male

2) **SELECT** statement in Hive is similar to the SELECT statement in SQL used for retrieving data from the database.

Syn: **SELECT * FROM tablename;** //displays all records

hive> SELECT * FROM employee.facultytable;

1	bob	25000.00	asstprof	35	male
2	mary	35000.00	assocprof	38	female
3	mike	50000.00	prof	45	male

SELECT col1,col2 FROM tablename; //Retrieves only specified columns data

hive> SELECT emp_name,emp_salary FROM employee.facultytable;

bob	25000.00
mary	35000.00
mike	50000.00

3) **a) INSERT INTO** statement appends the data into existing data in the table or partition.

Syn: **INSERT INTO TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...)] VALUES (col1value,col2value,...)**

hive> INSERT INTO TABLE employee.facultytable VALUES (4, 'jessy', 45000.00, 'assocprof', 40, 'female');

hive> SELECT * FROM employee.facultytable;

4	jessy	45000.00	assocprof	40	female
1	bob	25000.00	asstprof	35	male
2	mary	35000.00	assocprof	38	female
3	mike	50000.00	prof	45	male

b) INSERT OVERWRITE table overwrites the existing data in the table or partition.

Syn: **INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, ..) [IF NOT EXISTS]] select_statement FROM from_statement;**

4) DELETE statement in Hive deletes the table data. If the WHERE clause is specified, then it deletes the rows that satisfy the condition in where clause.

Syn: **DELETE FROM tablename [WHERE expression];**

hive> DELETE FROM employee.facultytable WHERE emp_age=38;

hive> SELECT * FROM employee.facultytable;

4	jessy	45000.00	assocprof	40	female
1	bob	25000.00	asstprof	35	male
3	mike	50000.00	prof	45	male

5) UPDATE statement in Hive deletes the table data. If the WHERE clause is specified, then it updates the column of the rows that satisfy the condition in WHERE clause. Partitioning and Bucketing columns cannot be updated.

Syn: **UPDATE tablename SET column = value [, column = value ...] [WHERE expression];**

hive> UPDATE employee.facultytable SET emp_name = 'mike tyson' WHERE emp_age=45;

hive> SELECT * FROM employee.facultytable;

4	jessy	45000.00	assocprof	40	female
1	bob	25000.00	asstprof	35	male
3	mike tyson	50000.00	prof	45	male

6) EXPORT statement exports the table or partition data along with the metadata to the specified output location in the HDFS. Metadata is exported in a _metadata file, and data is exported in a subdirectory 'data.'

Syn: **EXPORT TABLE tablename [PARTITION (part_column="value"[, ...])] TO 'export_target_path' [FOR replication('eventid')];**

hive> EXPORT TABLE employee.drivertable TO '/user/hive/warehouse';

7) IMPORT command imports the data from a specified location to a new table or already existing table.

Syn: **IMPORT [[EXTERNAL] TABLE new_or_original_tablename [PARTITION (part_column="value"[, ...])]] FROM 'source_path' [LOCATION 'import_target_path'];**

hive> IMPORT TABLE employee.importedtable FROM '/user/hive/warehouse';

Week-9 : Write a Map-Reduce program that mines weather data. Hint: Weather sensors collecting data every hour at many locations across the globe gather a large volume of log data, which is a good candidate for analysis with Map-Reduce, since it is semi-structured and record-oriented.

Step-1: Open Virtual box and then start **cloudera quickstart** Step-2: Open **Eclipse** present on the cloudera desktop Step-3: Creating Java Project

3.1 : File-> New -> Project -> **Java Project** -> Next ("MyMaxMin" is the Project name)

Step-4: Adding the Hadoop libraries to the Project

4.1 : Click on **Add External Jars** button, the, File File System > usr > lib> **Hadoop**, select all the libraray (jar) files and then click OK button

4.2 : Now Click on Add External Jars button, the, File File System > usr > lib> Hadoop>**client**, select all the libraray (jar) files and then click OK button

4.3 : Click **finish** button

Step-5: Create Java MapperReduce program

5.1 : In the explorer panel Right click on "src" folder of the project MyMaxMin New> Class> **Name** textfield give as "MyMaxMin" and click Finish button.

5.2 : Type the code for MyMaxMin program with import files, Mapper class, Reducer class Driver class with main method

Step-6: Export the project as JAR

6.1 : Right click MyMaxMin project and select "Export" >> Java >> Jar file >> Next>> in the JAR file textfield give as /home/cloudera/MyMaxMin.jar, click Finish button>> OK

6.2 : Open **cloudera@quickstart** terminal and verify the jar file using **ls** command Step-7: Create the input file for the MapReduce program by typing command

cat > /home/cloudera/inputFile.txt

Verify the data contents by **cat /home/cloudera/CRND 01 03 3 2015AK _ Fairbanks.txt**

Step-8: Move the input file created in local system to hdfs store by

hdfs dfs -put /home/cloudera/inputFile.txt /

view the contents of the file moved to hdfs by typing command

hdfs dfs -cat /CRND 01 03 3 -201 5-AK _ Fairbanks.txt

Step-9: Run MapReduce program on Hadoop by typing command

hadoop jar **/home/cloudera/My MaxMin.jar My MaxMin /CRND 01 03 3 -201 5- AK _ Fairbanks.txt /out25**

Each time you run the above command; you need to give different name for the output directory.

Step-10: View the output directory content by hdfs dfs -ls /out25 of the program/job executed,

hdfs dfs -cat /out25/part-r-00000

Example: Calculate Maximum and Minimum T emperature

```
// 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);


    }

}
```

i c touoe raequT cksoa rt - j s ndrs dos - put z nomes c i ouoe raw cncoo1oz - zO os -Ax i=a I rbanks

```
[cloudera@quickstart ~]$ hdfs dfs -ls /
Found 9 items
drwxr-xr-x   - cloudera supergroup          0 2021-03-30 02:53 /BDA-Lab
drwxr-xr-x   - cloudera supergroup          0 2021-03-30 02:50 /BDA-Lab1
-rw-r--r--   1 cloudera supergroup      79205 2021-04-05 21:32 /CRND0103-2015-AK_Fairbanks_11_NE.txt
drwxrwxrwx   - hdfs     supergroup          0 2017-10-23 09:15 /benchmarks
drwxr-xr-x   - hbase    supergroup          0 2021-04-05 20:54 /hbase
drwxr-xr-x   - solr     solr                0 2017-10-23 09:18 /solr
drwxrwxrwt   - hdfs     supergroup          0 2021-03-24 20:44 /tmp
drwxr-xr-x   - hdfs     supergroup          0 2017-10-23 09:17 /user
drwxr-xr-x   - hdfs     supergroup          0 2017-10-23 09:17 /var
[cloudera@quickstart ~]$ hadoop jar MyMaxMin.jar MyMaxMin /CRND0103-2015-AK Fairbanks 11 NE.txt /out25
```

2 tyfil4y fil5 2l : 3 4 : 52 WARN ma p red uce. J obRes ou r c eUpT oade r : Hadoop comma nd - l ine option pa rs ing not pe rfo rmed Implement the Tool interface and exec

```
21/04/05 21:34:52 INFO input.FileInputFormat: Total input paths to process : 1
21/04/05 21:34:52 INFO mapreduce.JobSubmitter: number of splits:1
21/04/05 21:34:52 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1617681225311_0001
21/04/05 21:34:53 INFO mapreduce.Job: The url to track the job: http://quickstart.cloudera:8088/proxy/application_1617681225311_0001/
21/04/05 21:34:53 INFO mapreduce.Job: Running job: job_1617681225311_0001
21/04/05 21:35:06 INFO mapreduce.Job: Job job_1617681225311_0001 running in uber mode : false
21/04/05 21:35:06 INFO mapreduce.Job:  map 0% reduce 0%
21/04/05 21:35:16 INFO mapreduce.Job:  map 100% reduce 0%
21/04/05 21:35:23 INFO mapreduce.Job:  map 100% reduce 100%
21/04/05 21:35:24 INFO mapreduce.Job: Job job_1617681225311_0001 completed successfully
21/04/05 21:35:24 INFO mapreduce.Job: Counters: 49
        File System Counters
                FILE: Number of bytes read=13422
```

File Machine View Input Devices Help
Applications Places System Mon Apr 5, 10:51 PM cloudera

```
                FILE. flum0er oI large read operations—0

                HDFS: Number of bytes read=79338
                HDFS: Number of bytes written=12690
                HDFS: Number of read operations=6
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
```

Total time spent by all reduces in occ upied slot s (ms) =392t

```
                loial vcore-milliseconds taken by all reduce tasks=39¥l

                Nap input records=365

                Reduce input resords=363

                Shuffled Maps =1
```

File Machine View Input Devices Help
Applications Places System Mon Apr 5, 10:52 PM cloudera

```
                Failed Shuffles=0

                GC time elapsed (ms)=155
                CPU time spent (ms)=1000
                Physical memory (bytes) snapshot=348438528
                Virtual memory (bytes) snapshot=3015147520
                Total committed heap usage (bytes)=226365440
        Shuffle Errors


                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=79205
        File Output Format Counters
                Bytes Written=12690
[cloudera@quickstart ~]$ hdfs dfs -ls /out25
Found 2 items
-rw-r--r--   1 cloudera supergroup          0 2021-04-05 21:35 /out25/_SUCCESS
-rw-r--r--   1 cloudera supergroup      12690 2021-04-05 21:35 /out25/part-r-00000
[cloudera@quickstart ~]$ hdfs dfs -cat /out25/part-r-00000
The Day is Cold Day :20150101   -9.7



The Day is Cold Day. 20150107   - 1.2 6
The Day is Cold Day :20150108  -10.3
The Day is Cold Day :20150109   -7.6
The Day is Cold Day :20150110   -8.2
```

Week-10	Pig commands – Pig Latin - Relational Operator: FILTER, SPLIT, WORDCOUNT program
10a	For the given Student dataset and Student attendance dataset, perform Relational operator – FILTER and SPLIT operations in Hadoop Pig framework using Cloudera

Student_data				
Student ID	**Name**	**Age**	**City**	**CGPA**
1	Jagruthi	21	Hyderabad	9.1
2	Praneeth	22	Chennai	8.6
3	Sujith	22	Mumbai	7.8
4	Sreeja	21	Bengaluru	9.2
5	Mahesh	24	Hyderabad	8.8
6	Rohit	22	Chennai	7.8
7	Sindhu	23	Mumbai	8.3

Step-1: **Create a Directory** in HDFS with the name **pigdir** in the required path using **mkdir**:

$ hdfs dfs -mkdir /bdalab/pigdir

Step-2: The input file of Pig contains each tuple/record in individual lines with the entities separated by a delimiter (“,”).

In the local file system, **create an input file student_data.txt** containing data as shown below.

1,Jagruthi,21,Hyderabad,9.1
2,Praneeth,22,Chennai,8.6
3,Sujith,22,Mumbai,7.8
4,Sreeja,21,Bengaluru,9.2
5,Mahesh,24,Hyderabad,8.8
6,Rohit,22,Chennai,7.8
7,Sindhu,23,Mumbai,8.3

Step-3: **Move the file** from the local file system to HDFS using **put (Or) copyFromLocal** command and verify using -cat command

$ hdfs dfs -put /home/cloudera/pigdir/student_data /bdalab/pigdir/

$ hdfs dfs -cat / bdalab/pigdir/student_data

Step-4: Open Pig in **Grunt shell** and execute the following Pig Latin statement.

Apply Relational Operator – LOAD to load the data into Relation name std_data from the file student_data.txt into Relational Operators are **NOT case sensitive.**

$ pig => will direct to **grunt>**

grunt> std_data =LOAD' /bdalab/pigdir/student_data' USING PigStorage(',') as (id:int, name:chararray, age:int, city:chararray, cgpa:double);

grunt> DUMP std_data;

(1,Jagruthi,21,Hyderabad,9.1)

```
(2,Praneeth,22,Chennai,8.6)
(3,Sujith,22,Mumbai,7.8)
(4,Sreeja,21,Bengaluru,9.2)
(5,Mahesh,24,Hyderabad,8.8)
(6,Rohit,22,Chennai,7.8)
(7,Sindhu,23,Mumbai,8.3)
```

Step-5: **FILTER** operator is used to select the required tuples from a relation based on a condition

```
grunt> filter_std = FILTER std_data BY city == 'Hyderabad';

grunt> DUMP filter_std;

(1,Jagruthi,21,Hyderabad,9.1)
(5,Mahesh,24,Hyderabad,8.8)
```

Step-6: **SPLIT** operator is used to split a relation into two or more relations

```
grunt> SPLIT std_data INTO split_std1 IF age<23, split_std2 IF (age>22 AND age<25);

grunt> DUMP split_std1;

(1,Jagruthi,21,Hyderabad,9.1)
(2,Praneeth,22,Chennai,8.6)
(3,Sujith,22,Mumbai,7.8)
(4,Sreeja,21,Bengaluru,9.2)
(6,Rohit,22,Chennai,7.8)

grunt> DUMP split_std2;

(5,Mahesh,24,Hyderabad,8.8)
(7,Sindhu,23,Mumbai,8.3)
```

10b	Develop a WordCount program using Pig Latin statements in Hadoop Pig framework using Cloudera

The input file of Pig contains each tuple/record in individual lines with the entities separated by a delimiter (",").

Step-1: **Create a Directory** in HDFS with the name **pigdir** in the required path using **mkdir**:

$ hdfs dfs -mkdir /bdalab/pigdir

Step-2: In the local file system, **create an input file wordcount** containing data as shown below.

Deer,Bear,River
Car,Car,River
River,Car,River
Deer,River,Bear

Step-3: **Move the file** from the local file system to HDFS using **put (Or) copyFromLocal** command and verify using -cat command

$ hdfs dfs -put /home/cloudera/pigdir/wordcount_data /bdalab/pigdir/

$ hdfs dfs -cat / bdalab/pigdir/ wordcount_data

Step-4: Open Pig in **Grunt shell** and execute the following Pig Latin statement.

$ pig => will direct to **grunt>**

Convert Each line to each tuple.

Apply Relational Operator – **LOAD** to load the data into Relation **lines** from the file **wordcount_data.**

```
grunt> lines = LOAD '/bdalab/pigdir/wordcount_data' AS (line:chararray);
```

```
grunt> DUMP lines;
```

(Deer,Bear,River)
(Car,Car,River)
(River,Car,River)
(Deer,River,Bear)

Step-5: **Convert Each line tuple to each word tuple**

TOKENIZE splits the line into a field for each word.

FLATTEN will take the collection of records returned by TOKENIZE and produce a separate record for each one, calling the single field in the record word.

FOREACH operator is used to generate specified data transformations based on the column data.

```
grunt> words = FOREACH lines GENERATE FLATTEN(TOKENIZE(line)) as word;
```

```
grunt> dump words;
```

(Deer)
(Bear)
(River)

(Car)
(Car)
(River)
(River)
(Car)
(River)
(Deer)
(River)
(Bear)

Step-6: **Group all similar words into each tuple**

```
grunt> groupword = GROUP words by word;
grunt> dump groupword;
(Car,{(Car),(Car),(Car)})
(Bear,{(Bear),(Bear)})
(Deer,{(Deer),(Deer)})
(River,{(River),(River),(River),(River),(River)})
```

Step-6: **Count each grouped word and display**

```
grunt> wordcount = FOREACH groupword GENERATE group, COUNT(words);
grunt> dump wordcount;
(Car,3)
(Bear,2)
(Deer,2)
(River,5)
```

PART-3

ONLINE RESOURCES

ONLINE RESOURCES

OBJECTIVE

To help students on acquiring more practice on the course using various online resources

DESCRIPTION

These open tools form an excellent practice platform for the students, which he can explore anytime from anywhere. The links of the websites providing Big data tutorials are given. Students are needed to explore the websites.

LINKS

JOURNALS/MAGAZINES

http://www.sciepub.com/journal/AJSE

*

SWAYAM/NPTEL/MOOCs:

https://nptel.ac.in/courses/106/105/106105167/

PART-4

POSSIBLE VIVA QUESTIONS

POSSIBLE VIVA QUESTIONS

Description

Possible viva questions includes, questions whose answers are provided. The student can have a practice of them.

Socures: https://www.javatpoint.com/cloud-computing-interview-questions

1. **What do you understand by the term 'big data'?**

Big data deals with complex and large sets of data that cannot be handled using conventional software.

2. **How is big data useful for businesses?**

Big Data helps organizations understand their customers better by allowing them to draw conclusions from large data sets collected over the years. It helps them make better decisions.

3. **What is the Port Number for NameNode?**

NameNode - Port 50070

4. **What is the function of the JPS command?**

The JPS command is used to test whether all the Hadoop daemons are running correctly or not.

5. **What is the command to start up all the Hadoop daemons together?**

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

6. **Name a few features of Hadoop.**

Some of the most useful features of Hadoop,

1. It's open sourcenature.
2. User-friendly.
3. Scalability.
4. Data locality.
5. Data recovery.

7. **What are the five V's of Big Data?**

The five V's of Big data are Volume, Velocity, Variety, Veracity, and Value.

8. **What are the components of HDFS?**

The two main components of HDFS are:

1. Name Node
2. Data Node

9. **How is Hadoop related to Big Data?**

Hadoop is a framework that specializes in big data operations.

10. **Name a few data management tools used with Edge Nodes?**

Oozie, Flume, Ambari, and Hue are some of the data management tools that work with edge nodes in Hadoop.

11. **What are the steps to deploy a Big Data solution?**

The three steps to deploying a Big Data solution are:

1. Data Ingestion
2. Data Storage and
3. Data Processing

12. **How many modes can Hadoop be run in?**

Hadoop can be run in three modes— Standalone mode, Pseudo-distributed mode and fully-distributed mode.

13. **Name the core methods of a reducer**

The three core methods of a reducer are,

1. setup()
2. reduce()
3. cleanup()

14. **What is the command for shutting down all the Hadoop Daemons together?**

```
./sbin/stop-all.sh
```

15. **What is the role of NameNode in HDFS?**

NameNode is responsible for processing metadata information for data blocks within HDFS.

16. **What is FSCK?**

FSCK (File System Check) is a command used to detect inconsistencies and issues in the file.

17. **What are the real-time applications of Hadoop?**

Some of the real-time applications of Hadoop are in the fields of:

- Content management.
- Financial agencies.
- Defense and cybersecurity.

- Managing posts on social media.

18. **What is the function of HDFS?**

The HDFS (Hadoop Distributed File System) is Hadoop's default storage unit. It is used for storing different types of data in a distributed environment.

19. **What is commodity hardware?**

Commodity hardware can be defined as the basic hardware resources needed to run the Apache Hadoop framework.

20. **Name a few daemons used for testing JPS command.**

- NameNode
- NodeManager
- DataNode
- ResourceManager

21. **What are the most common input formats in Hadoop?**

- Text Input Format
- Key Value Input Format
- Sequence File Input Format

22. **Name a few companies that use Hadoop.**

Yahoo, Facebook, Netflix, Amazon, and Twitter.

23. **What is the default mode for Hadoop?**

Standalone mode is Hadoop's default mode. It is primarily used for debugging purpose.

24. **What is the role of Hadoop in big data analytics?**

By providing storage and helping in the collection and processing of data, Hadoop helps in the analytics of big data.

25. **What are the components of YARN?**

The two main components of YARN (Yet Another Resource Negotiator) are:

- Resource Manager
- Node Manager

PART-5

KNOWLEDGE BASE

KNOWLEDGE BASE

Description

Knowledge base includes possible viva questions whose answers are not provided. The student has to explore for the answers.

POSSIBLE VIVA QUESTIONS (WITHOUT ANSWERS)

1. Why BigData use of BIgData?
2. Hadoop eco system.
3. Use of HDFS, What are name nodes and data nodes.
4. Use of Map Reduce in Hadoop.
5. What are 3 v's in big data?
6. HDFS Commands: creating files, adding files, deleting files and retrieving files.
7. What is FS data input stream, FS data output stream in HDFS.
8. What is Hive, Pig and Mongodb in hadoop Ecosystem?