Centenary Celebrated Sharnbasveshwar Vidya Vardhaka Sangha's

ಶರಣಬಸವ ವಿಶ್ವವಿದ್ಯಾಲಯ

SHARNBASVA UNIVERSITY

A State Private University approved by Govt. of Karnataka vide Notification No. ED 144 URC 2016 dated 29-07-2017
Recognised by UGC under Section 2f vide No. F.8-29/2017 (CPP-I/PU), dated 20-12-2017 & AICTE, CoA, PCI New Delhi

DEPARTMENT OF ARTIFICIAL INTELLIGENCE & DATA SCIENCE

FACULTY OF ENGINEERING AND TECHNOLOGY(CO-ED)

SHARNBASVA UNIVERSITY KALABURAGI

BIG DATA ANALYTICS

Lab Manual

For 6th Semester

Course Code: 22ADL67

By

Dr. Laxmi Math

List of Experiments

Sl. NO	Title of the Experiment
1	Install Hadoop and Implement the following file management tasks in Hadoop: Adding files and directories,Retrieving files,Deleting files and directories. **Hint:** A typical Hadoop workflow creates data files (such as log files) elsewhere and copies them into HDFS using one of the above command line utilities.
2	Develop a MapReduce program to implement Matrix Multiplication
3	Develop a Map Reduce program that mines weather data and displays appropriate messages indicating the weather conditions of the day.
4	Develop a MapReduce program to find the tags associated with each movie by analyzing movie lens data.
5	Implement Functions: Count – Sort – Limit – Skip – Aggregate using MongoDB
6	Develop Pig Latin scripts to sort, group, join, project, and filter the data.
7	Use Hive to create, alter, and drop databases, tables, views, functions, and indexes.
8	Implement a word count program in Hadoop and Spark.
9	Use CDH (Cloudera Distribution for Hadoop) and HUE (Hadoop User Interface) to analyze data and generate reports for sample datasets

1. Install Hadoop and Implement the following file management tasks in Hadoop: Adding files and directories,Retrieving files,Deleting files and directories.

Hint: A typical Hadoop workflow creates data files (such as log files) elsewhere and copies them into HDFS using one of the above command line utilities.

Lab: Installing Hadoop and Implementing File Management Tasks in Ubuntu

This lab will guide you through installing Hadoop on Ubuntu and performing basic file management tasks using Hadoop Distributed File System (HDFS), such as adding files and directories, retrieving files, and deleting files and directories.

2. Step-by-Step Guide

Step 1: Install Java

Hadoop requires Java to run. You can install Java 8 or 11 by running the following commands:

sudo apt update

sudo apt install openjdk-8-jdk

Verify Java installation:

java -version

The output should show Java 8 or 11 version, depending on which you installed.

Step 2: Download and Install Hadoop

Download the latest stable version of Hadoop from the official Apache Hadoop website.

wget https://downloads.apache.org/hadoop/common/stable/hadoop-3.3.3.tar.gz

Extract the downloaded file:

tar -xzvf hadoop-3.3.3.tar.gz

Move the extracted files to the /usr/local directory:

sudo mv hadoop-3.3.3 /usr/local/hadoop

Set up environment variables to configure Hadoop:

Open the .bashrc file in a text editor:

nano ~/.bashrc

Add the following lines at the end of the file:

export HADOOP_HOME=/usr/local/hadoopexport
PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbinexport
HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoopexport

HADOOP_MAPRED_HOME=$HADOOP_HOMEexport HADOOP_COMMON_HOME=$HADOOP_HOMEexport HADOOP_HDFS_HOME=$HADOOP_HOMEexport YARN_HOME=$HADOOP_HOMEexport JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64

Source the .bashrc to apply the changes:

source ~/.bashrc

Verify Hadoop is installed correctly:

hadoop version

Step 3: Configure Hadoop

Open the Hadoop configuration directory:

cd /usr/local/hadoop/etc/hadoop

Edit core-site.xml to configure the Hadoop file system (HDFS) settings:

nano core-site.xml

Add the following configuration inside the <configuration> tags:

xml

CopyEdit

```
<property>
   <name>fs.defaultFS</name>
   <value>hdfs://localhost:9000</value></property>
```

Edit hdfs-site.xml to configure HDFS storage:

nano hdfs-site.xml

Add the following configuration inside the <configuration> tags:

xml

CopyEdit

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

<name>dfs.namenode.name.dir</name>

<value>file:///usr/local/hadoop/hdfs/name</value></property>

<property>

<name>dfs.datanode.data.dir</name>

<value>file:///usr/local/hadoop/hdfs/data</value></property>

Edit mapred-site.xml to configure MapReduce:

cp mapred-site.xml.template mapred-site.xml

nano mapred-site.xml

Add the following configuration:

xml

CopyEdit

<property>

<name>mapreduce.framework.name</name>

<value>yarn</value></property>

Edit yarn-site.xml to configure YARN:

nano yarn-site.xml

Add the following configuration:

xml

CopyEdit

<property>

<name>yarn.resourcemanager.hostname</name>

<value>localhost</value></property>

Step 4: Format the Hadoop NameNode

Before starting Hadoop, you need to format the Hadoop NameNode:

hdfs namenode -format

Step 5: Start Hadoop Daemons

Start the Hadoop daemons for HDFS and YARN:

Start HDFS (NameNode and DataNode):

start-dfs.sh

Start YARN (ResourceManager and NodeManager):

start-yarn.sh

Check if all daemons are running:

jps

You should see processes like NameNode, DataNode, ResourceManager, and NodeManager.

6. HDFS File Management Tasks

Now that Hadoop is set up, let's perform basic file management tasks using HDFS.

Task 1: Adding Files and Directories to HDFS

1.Create a directory in HDFS:

2.hadoop fs -mkdir /user/hadoop/input

1.Copy a local file to HDFS:

2.hadoop fs -put localfile.txt /user/hadoop/input/

Task 2: Retrieving Files from HDFS

Copy a file from HDFS to the local file system:

hadoop fs -get /user/hadoop/input/localfile.txt /home/ubuntu/

Task 3: Deleting Files and Directories in HDFS

Delete a file in HDFS:

hadoop fs -rm /user/hadoop/input/localfile.txt

Delete a directory recursively:

hadoop fs -rm -r /user/hadoop/input/

7. Verifying File Management Tasks

To list files in a directory in HDFS:

hadoop fs -ls /user/hadoop/input/

To check if a file exists:

hadoop fs -test -e /user/hadoop/input/localfile.txt

If the file exists, it will return 0, else it will return 1.

8. Stopping Hadoop Daemons

After completing your tasks, stop the Hadoop daemons:

Stop HDFS:

stop-dfs.sh

Stop YARN:

stop-yarn.sh

2. Develop a MapReduce program to implement Matrix Multiplication

```
from pyspark import SparkContext
sc = SparkContext("local", "MatrixMultiply")
# Read files
linesA = sc.textFile("matrix_a.txt")
linesB = sc.textFile("matrix_b.txt")
# Parse input
def parse_matrix_A(line):
    _, i, j, value = line.split(',')
    return (int(i), int(j), float(value))
def parse_matrix_B(line):
    _, j, k, value = line.split(',')
    return (int(j), int(k), float(value))
matrixA = linesA.map(parse_matrix_A)
matrixB = linesB.map(parse_matrix_B)
# Emit for multiplication: key = j (shared dimension)
# A: (i, j, a_ij) => (j, ('A', i, a_ij))
# B: (j, k, b_jk) => (j, ('B', k, b_jk))
mappedA = matrixA.map(lambda x: (x[1], ('A', x[0], x[2])))
mappedB = matrixB.map(lambda x: (x[0], ('B', x[1], x[2])))
# Join on j
joined = mappedA.union(mappedB).groupByKey()
# For each (j, list), compute partial products
def compute_products(_, values):
    A_values = []
    B_values = []
```

```
    for v in values:
        if v[0] == 'A':
            A_values.append((v[1], v[2]))  # (i, a_ij)
        else:
            B_values.append((v[1], v[2]))  # (k, b_jk)
    for i, a in A_values:
        for k, b in B_values:
            yield ((i, k), a * b)
products = joined.flatMap(lambda kv: compute_products(kv[0], kv[1]))
# Sum partial results
result = products.reduceByKey(lambda x, y: x + y)
# Display result
for ((i, k), value) in result.collect():
    print(f"C[{i}][{k}] = {value}")
```

INPUT FILES

1ST FILE SAVE AS matrix_a.txt

A,0,0,1

A,0,1,2

A,1,0,3

A,1,1,4

2nd FILE SAVE AS matrix_b.txt

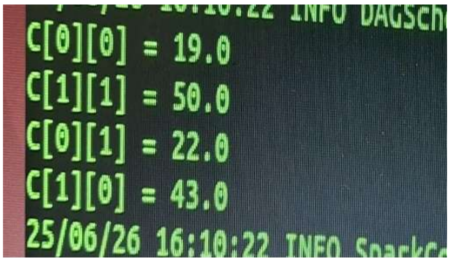
B,0,0,5

B,0,1,6

B,1,0,7

B,1,1,8

OUTPUT: cmd:- spark-submit filename.py



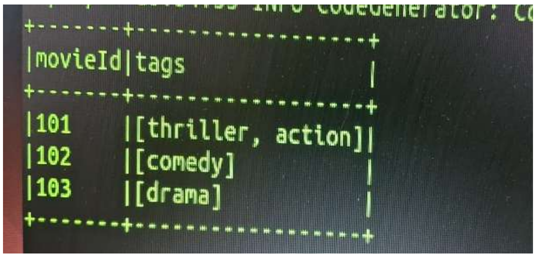
3. Develop a Map Reduce program that mines weather data and displays appropriate messages indicating the weather conditions of the day.

```
from pyspark.sql import SparkSession

from pyspark.sql.functions import collect_set

# Step 1: Start Spark

spark = SparkSession.builder.appName("MovieTags").getOrCreate()

# Step 2: Read CSV file

df = spark.read.csv("movietags.csv", header=True)

# Step 3: Select movieId and tag columns only

df_tags = df.select("movieId", "tag").dropDuplicates()

# Step 4: Group by movieId and collect all unique tags

movie_tags = df_tags.groupBy("movieId").agg(collect_set("tag").alias("tags"))

# Step 5: Show result

movie_tags.show(truncate=False)

# Step 6: Stop Spark

spark.stop()
```

INPUT FILE USE BELOW DATASET Save as .csv

userId	movieId	tag	timestamp
1	101	action	1430164910
2	101	thriller	1430164980
1	102	comedy	1430164990
3	101	action	1430165000
2	103	drama	1430165050

OUTPUT



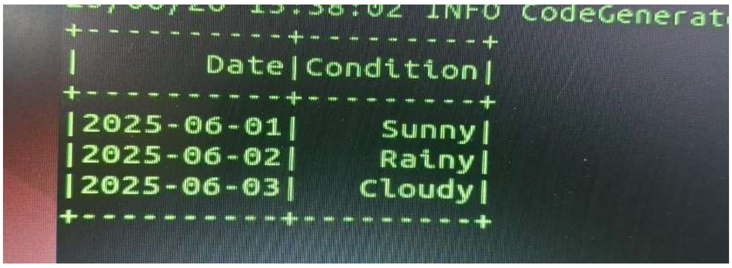
4. Develop a MapReduce program to find the tags associated with each movie by analyzing movie lens data.

```
from pyspark.sql import SparkSession

from pyspark.sql.functions import count

# Start Spark

spark = SparkSession.builder.appName("SimpleWeather").getOrCreate()

# Load CSV

df = spark.read.csv("weather.csv", header=True, inferSchema=True)

# Count each condition per day

condition_counts = df.groupBy("Date", "Condition").agg(count("*").alias("count"))

# Sort so most frequent conditions are on top

sorted_counts = condition_counts.orderBy("Date", "count", ascending=[True, False])

# Drop duplicate dates, keeping the first (most frequent)

top_conditions = sorted_counts.dropDuplicates(["Date"])

# Show result

top_conditions.select("Date", "Condition").show()

# Stop Spark

spark.stop()
```

INPUT FILE DATASET Save as .csv

Date	Temperature	Condition
2025-06-01	30	Sunny
2025-06-01	32	Sunny
2025-06-01	31	Sunny
2025-06-02	22	Rainy
2025-06-02	23	Cloudy
2025-06-02	21	Rainy
2025-06-03	18	Cloudy
2025-06-03	19	Cloudy
2025-06-03	20	Sunny

OUTPUT


+----------+---------+
| Date|Condition|
+----------+---------+
|2025-06-01| Sunny|
|2025-06-02| Rainy|
|2025-06-03| Cloudy|
+----------+---------+

5. Implement Functions: Count – Sort – Limit – Skip – Aggregate using MongoDB

Create a file :filename.js

```
 const { MongoClient } = require('mongodb');
const uri = 'mongodb://localhost:27017'; // Update if using MongoDB Atlas
const client = new MongoClient(uri);
async function run() {
  try {
    await client.connect();
    const db = client.db('testDB');
    const users = db.collection('users');
    // 1. Count
    const count = await users.countDocuments({ city: "New York" });
    console.log("Count of users from New York:", count);

    // 2. Sort (by age descending)
    const sortedUsers = await users.find().sort({ age: -1 }).toArray();
    console.log("Sorted users by age (desc):", sortedUsers);

    // 3. Limit (get top 2)
    const limitedUsers = await users.find().limit(2).toArray();
    console.log("Top 2 users:", limitedUsers);

    // 4. Skip (skip first 2)
    const skippedUsers = await users.find().skip(2).toArray();
    console.log("Users after skipping first 2:", skippedUsers);
```

```
    // 5. Aggregate (group by city and count users)
    const aggregation = await users.aggregate([
      { $group: { _id: "$city", count: { $sum: 1 } } },
      { $sort: { count: -1 } }
    ]).toArray();
    console.log("User count by city:", aggregation);

  } finally {
    await client.close();
  }
}
run().catch(console.error);
```

INPUT:

Step 1: Insert Sample Data into MongoDB

>mongosh

Then paste:

```
use testDB
db.users.insertMany([
  { name: "Alice", age: 25, city: "New York" },
  { name: "Bob", age: 30, city: "Los Angeles" },
  { name: "Charlie", age: 35, city: "Chicago" },
  { name: "David", age: 28, city: "New York" },
  { name: "Eve", age: 22, city: "Chicago" }
])
```

STEP 2: RUN THE PGM

node filename.js

OUTPUT:

Total users: 5

Sorted by age: [{ name: 'Charlie', age: 22, ... }, ...]

Limit 2 users: [{ name: 'Alice', ... }, { name: 'Bob', ... }]

Skip 2 users: [{ name: 'Charlie', ... }, { name: 'David', ... }, ...]

Group by city: [{ _id: 'New York', total: 2 }, { _id: 'London', total: 2 }, { _id: 'Paris', total: 1 }]

6. Develop Pig Latin scripts to sort, group, join, project, and filter the data.

INPUT:

a. students.txt

1,John,22

2,Alice,23

3,Bob,22

4,David,24

Each line: **StudentID,Name,Age**

b. marks.txt

1,85

2,91

3,78

4,88

Each line: **StudentID,Marks**

PROGRAM:

```
-- Load the files
A = LOAD 'students.txt' USING PigStorage(',') AS (id:int, name:chararray, age:int);
B = LOAD 'marks.txt' USING PigStorage(',') AS (id:int, marks:int);

-- Filter students older than 22
C = FILTER A BY age > 22;

-- Select only name and age
D = FOREACH C GENERATE name, age;

-- Join student info with marks
E = JOIN A BY id, B BY id;
```

```
-- Group by age
F = GROUP A BY age;

-- Count students in each age group
G = FOREACH F GENERATE group AS age, COUNT(A) AS total;

-- Sort by age descending
H = ORDER A BY age DESC;

-- Save output
STORE D INTO 'out_filtered' USING PigStorage(',');
STORE E INTO 'out_joined' USING PigStorage(',');
STORE G INTO 'out_grouped' USING PigStorage(',');
STORE H INTO 'out_sorted' USING PigStorage(',');
```

OUTPUT: pig -x local student_analysis.pig

✅ How to Run It

Step 1: Save input files in the current folder

```bash
echo -e "1,John,22\n2,Alice,23\n3,Bob,22\n4,David,24" > students.txt
echo -e "1,85\n2,91\n3,78\n4,88" > marks.txt
```

Step 2: Save the Pig script

```bash
nano student.pig
```

Paste the script above, then save.

Step 3: Run Pig in local mode

```bash
pig -x local student.pig
```

7. Use Hive to create, alter, and drop databases, tables, views, functions, and indexes.

▶️ How to Run It

1. Save it as a file, e.g. `hive_script.sql`.
2. Open Hive shell:

```bash
hive
```

3. Run the script:

```sql
SOURCE /full/path/to/hive_script.sql;
```

-- Step 1: Create Database

CREATE DATABASE mydb;

-- Step 2: Use the Database

USE mydb;

-- Step 3: Alter Database Properties

ALTER DATABASE mydb SET DBPROPERTIES ('edited-by' = 'hive-user');

-- Step 4: Create Table

```
CREATE TABLE employees (
   id INT,
   name STRING,
   salary FLOAT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

```
-- Step 5: Alter Table - Add Column
ALTER TABLE employees ADD COLUMNS (age INT);

-- Step 6: Rename Table
ALTER TABLE employees RENAME TO workers;

-- Step 7: Create View
CREATE VIEW high_salary AS
SELECT * FROM workers WHERE salary > 5000;

-- Step 8: Recreate View (simulating ALTER)
DROP VIEW high_salary;
CREATE VIEW high_salary AS
SELECT * FROM workers WHERE salary > 10000;

-- Step 9: Register Temporary UDF (adjust path/classname if needed)
-- NOTE: This requires the JAR file to exist. Uncomment if you have a UDF.
-- ADD JAR /path/to/myfunc.jar;
-- CREATE TEMPORARY FUNCTION my_upper AS 'com.example.MyUpperFunction';

-- Step 10: Create Index
CREATE INDEX emp_index ON TABLE workers (id)
AS 'COMPACT'
WITH DEFERRED REBUILD;

-- Step 11: Rebuild Index
ALTER INDEX emp_index ON workers REBUILD;
```

```
-- Step 12: Drop Index
DROP INDEX emp_index ON workers;

-- Step 13: Drop View
DROP VIEW high_salary;

-- Step 14: Drop Table
DROP TABLE workers;

-- Step 15: Drop Database
DROP DATABASE mydb CASCADE;
```

8. Implement a word count program in Hadoop and Spark.

```
from pyspark import SparkContext
sc = SparkContext("local", "WordCount")
# Load file
text_file = sc.textFile("input8.txt")
# Word count logic
counts = text_file.flatMap(lambda line: line.split()) \
              .map(lambda word: (word, 1)) \
              .reduceByKey(lambda a, b: a + b)
# Print result
for word, count in counts.collect():
   print(f"{word}: {count}")
```

INPUT FILE Save as .csv

Virat is talking with ABD

ABD is listening

OUTPUT