Ex No: 1 Installation of Hadoop Framework,

AIM:

Installation of Hadoop Framework, it's components and study the HADOOP ecosystem

Hadoop is an open-source framework that allows to store and process big data in a distributed environment across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage.

Hadoop Architecture:

The Apache Hadoop framework includes following four modules:

Hadoop Common:

Contains Java libraries and utilities needed by other Hadoop modules. These libraries give file system and OS level abstraction and comprise of the essential Java files and scripts that are required to start Hadoop.

Hadoop Distributed File System (HDFS): A distributed file-system that provides highthroughput access to application data on the community machines thus providing very high aggregate bandwidth across the cluster.

Hadoop YARN: A resource-management framework responsible for job scheduling and cluster resource management.

Hadoop MapReduce: This is a YARN- based programming model for parallel processing of large data sets.

Hadoop Installation procedure:

Step 1: Download and install Java

https://www.oracle.com/java/technologies/javase-downloads.html

Step 2: Download Hadoop

https://hadoop.apache.org/releases.html

Step 3: Set Environment Variables

Step 4: Setup Hadoop

ou must configure Hadoop in this phase by modifying several configuration files. Navigate to the "etc/hadoop" folder in the Hadoop folder. You must make changes to three files:

core-site.xml

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

hdfs-site.xml

```
<configuration>
 <property>
 <name>dfs.replication</name>
 <value>1</value>
 </property>
 <property>
 <name>dfs.namenode.name.dir</name>
 <value>file:/hadoop-3.3.1/data/namenode</value>
 </property>
 <property>
 <name>dfs.datanode.data.dir</name>
 <value>file:/hadoop-3.3.1/data/datanode</value>
 </property>
</configuration>
```

mapred-site.xml

```
<configuration>
 <property>
 <name>mapred.job.tracker</name>
 <value>localhost:54311</value>
 </property>
</configuration>
```

Step 5: Format Hadoop NameNode `hadoop namenode -format`

Step 6: Start Hadoop

```
start-all.cmd
```

Step 7: Verify Hadoop

Installation

```
http://localhost:50070/.
```

Ex No: 2 File Management tasks in Hadoop

AIM:

To perform various file operation in HDFS

Step 1: Adding Files and Directories to HDFS

Before running Hadoop programs on data stored in HDFS, the data needs to be added to HDFS. Let's start by creating a directory and adding a file to it.

1. **Create a directory in HDFS:**

   `hadoop fs -mkdir /user/myfile`

   This command creates a new directory named `myfile` in the `/user` directory in

HDFS. 2. **Add a file to HDFS:**

   `hadoop fs -put a.txt`

   This command uploads the file `a.txt` from the local filesystem to the root directory of HDFS.

3. **Add the file to the newly created directory:**

   `hadoop fs -put a.txt /user/myfile`

   This command uploads the file `a.txt` from the local filesystem directly into the `/user/myfile` directory in HDFS.

Step 2: Retrieving Files from HDFS

To copy files from HDFS back to the local filesystem, use the `get` command. Here's how to retrieve `a.txt`:

`hadoop fs -cat a.txt`

This command displays the contents of the file a.txt directly to the console. To actually copy the file to the local filesystem, you would use:

hadoop fs -get a.txt /local/path

Replace /local/path with the desired path on your local filesystem.

Step 3: Deleting Files from HDFS

To delete a file from HDFS, use the rm command. Here's how to delete

a.txt: hadoop fs -rm a.txt

This command removes the file a.txt from HDFS.

Output

The successful execution of the above commands will result in the following:

- Creation of the /user/myfile directory in HDFS.
- Addition of a.txt to HDFS and then to /user/myfile.
- Retrieval of a.txt from HDFS to the local filesystem.
- Deletion of a.txt from HDFS.

Result

The program of file management tasks in Hadoop has been executed successfully, and the output has been verified

Ex No: 3 Implement word count program using Map Reduce.

AIM:

To implementing distinct word count problem using Map-Reduce

Algorithm :

The function of the mapper is as follows:

- Create a IntWritable variable 'one' with value as 1
- Convert the input line in Text type to a String
- Use a tokenizer to split the line into words
- Iterate through each word and a form key value pairs as Assign each work from the tokenizer (of String type) to a Text 'word'
- Form key value pairs for each word as < word,one > and push it to the output collector

The function of Sort and Group:

- After this, "aggregation" and "Shuffling and Sorting" done by framework. Then Reducers task these final pair to produce output.

The function of the reducer is as follows

- Initialize a variable 'sum' as 0
- Iterate through all the values with respect to a key and sum up all of them • Push

to the output collector the Key and the obtained sum as value For Example:

Example:

For the given sample input1 data file (input1.txt : Hello World Bye World) mapper emits: <Hello,1>

<World,1>

<Bye,1>

<World,1>

The second input2 data file (input2.txt : Hello Hadoop Goodbye Hadoop) mapper emits: <Hello,1>

<Hadoop,1>

<Goodbye,1>

<Hadoop,1>

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

The output of the first map:

<Hello,1>

<Bye,1>

<World,2>

The output of the second map:

<Hello,1>

<Hadoop,2>

<Goodbye,1>

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

Thus the output of the job is:

<Goodbye,1>

<Bye,1>

<Hello,2>

<Hadoop,2>

<World,2>

Python Implementation

mapper.py

```
#!/usr/bin/env python3
```

```
import sys

# Mapper Function
for line in sys.stdin:
    line = line.strip()              # Remove leading/trailing spaces
    words = line.split()             # Tokenize the line into words
    for word in words:
        print(f"{word}\t1")          # Emit key-value pair <word,1>
```

reducer.py

```
#!/usr/bin/env python3
import sys

current_word = None
current_count = 0
word = None

# Reducer Function
for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    try:
        count = int(count)
    except ValueError:
        continue

    if current_word == word:
        current_count += count
    else:
        if current_word:
            print(f"{current_word}\t{current_count}")
        current_word = word
        current_count = count

# Emit the last word
if current_word == word:
    print(f"{current_word}\t{current_count}")
```

To Run on Hadoop :

```
hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-*.jar \
-input input_data/ \
-output output_data/ \
-mapper mapper.py \
-reducer reducer.py
```

Expected Output:

```
Bye     1
Goodbye 1
Hadoop  2
Hello   2
World   2
```

Result:

Thus, the Word Count program using MapReduce was successfully implemented and executed to count the occurrences of each distinct word from multiple input files.

Ex No: 4 Weather Report POC using Hadoop Streaming

AIM:

To write a Hadoop Streaming MapReduce program in Python to analyze weather data and generate a report containing maximum and minimum temperatures per day.

Algorithm :

Mapper Algorithm
1. Read a line from input.
2. Split the line into datetime and temperature.
3. Extract date from datetime.
4. Emit (date, temperature).

Reducer Algorithm
1. Receive (date, list of temperatures) from all mappers.
2. Track current date and temperature values.
3. When date changes, output max and min temperature for previous date.
4. After loop ends, output max and min for the last date.

Python Implementation

Mapper (mapper.py)

```
#!/usr/bin/env python3
import sys

# Input format: "datetime,temp"
# Example: "2025-09-01 14:00,35"

for line in sys.stdin:
    try:
        line = line.strip()
        datetime, temp = line.split(",")
        date = datetime.split(" ")[0]  # extract only date
        print(f"{date}\t{temp}")
    except:
        continue  # skip malformed lines
```

Reducer (reducer.py)

```
#!/usr/bin/env python3
import sys

current_date = None
temps = []

for line in sys.stdin:
    line = line.strip()
```

```
    if not line:
        continue
    date, temp = line.split("\t")
    temp = float(temp)

    if current_date == date:
        temps.append(temp)
    else:
        if current_date:
            # output result for the previous date
            print(f"{current_date}\tmax={max(temps)}\tmin={min(temps)}")
        current_date = date
        temps = [temp]

# Final output for the last date
if current_date:
    print(f"{current_date}\tmax={max(temps)}\tmin={min(temps)}")
```

Sample Input (weather_data.txt)

```
2025-09-01 14:00,35
2025-09-01 15:00,33
2025-09-01 16:00,37
2025-09-02 14:00,32
2025-09-02 15:00,34
```

Running the Program in Hadoop Streaming

```
hadoop jar /path/to/hadoop-streaming.jar \
    -input /user/hadoop/weather_data.txt \
    -output /user/hadoop/weather_output \
    -mapper mapper.py \
    -reducer reducer.py \
    -file mapper.py \
    -file reducer.py
```

Sample Output

```
2025-09-01    max=37.0    min=33.0
2025-09-02    max=34.0    min=32.0
```

Result

The Hadoop Streaming MapReduce program was successfully executed to generate a daily weather report showing maximum and minimum temperatures.

Ex No: 5 Implement Pig Latin scripts to sort, group, join, project, and filter your data

AIM:

To implement Pig Latin scripts to load, filter, project, group, sort, and join datasets using Apache Pig.

Algorithm :

1. **Load the Data**
   Use LOAD command to read data from CSV files using PigStorage(',').
   Define schema (column names and types).
2. **Filter Operation**
   Use FILTER to select tuples based on a condition (e.g., marks > 60).
3. **Projection Operation**
   Use FOREACH ... GENERATE to select specific columns.
4. **Group Operation**
   Use GROUP to group tuples by a particular field (e.g., department).
5. **Sort Operation**
   Use ORDER BY to sort tuples in ascending or descending order.
6. **Join Operation**
   Use JOIN to combine two datasets on a common key (e.g., department).
7. **Display Results**
   Use DUMP to display intermediate and final results.

Example Input Files

students.csv

1,Ravi,CSE,85
2,Anita,IT,55
3,John,CSE,72
4,Kiran,ECE,67
5,Meera,IT,90

departments.csv

CSE,Dr.Sharma
IT,Dr.Verma
ECE,Dr.Rao

Python Implementation

```
!wget  https://downloads.apache.org/pig/pig-0.17.0/pig-0.17.0.tar.gz
!tar -xzf pig-0.17.0.tar.gz
!mv pig-0.17.0 /content/pig
```

```
import os
os.environ['PIG_HOME'] = '/content/pig'
os.environ['PATH'] += os.pathsep + os.path.join(os.environ['PIG_HOME'], 'bin')

# ================================
# 2. Create Input CSV Files
# ================================
students = """1,Ravi,CSE,85
2,Anita,IT,55
3,John,CSE,72
4,Kiran,ECE,67
5,Meera,IT,90
"""
with open("students.csv", "w") as f:
    f.write(students)

departments = """CSE,Dr.Sharma
IT,Dr.Verma
ECE,Dr.Rao
"""
with open("departments.csv", "w") as f:
    f.write(departments)

# ================================
# 3. Write the Pig Latin Script
# ================================
pig_script = r"""
-- Load student and department data
students = LOAD 'students.csv' USING PigStorage(',')
           AS (id:int, name:chararray, dept:chararray, marks:int);

departments = LOAD 'departments.csv' USING PigStorage(',')
              AS (dept:chararray, hod:chararray);

-- Filter: select students with marks > 60
good_students = FILTER students BY marks > 60;

-- Project: select only name, dept, marks
projected = FOREACH good_students GENERATE name, dept, marks;

-- Group: group by department
grouped = GROUP projected BY dept;

-- Sort: order by marks descending
sorted = ORDER projected BY marks DESC;

-- Join: combine students with department HODs
joined = JOIN projected BY dept, departments BY dept;

-- Dump results
DUMP sorted;
DUMP grouped;
```

```
DUMP joined;
"""

with open("program.pig", "w") as f:
    f.write(pig_script)

# ================================
# 4. Set Java Environment & Run Pig Script (Local Mode)
# ================================
!export  JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
!export PATH=$JAVA_HOME/bin:$PATH

os.environ['JAVA_HOME'] = '/usr/lib/jvm/java-11-openjdk-amd64'
os.environ['PATH'] = os.environ['JAVA_HOME'] + '/bin:' + os.environ['PATH']

!pig -x local program.pig
```

Expected Output:

Sorted Output

(Meera,IT,90)
(Ravi,CSE,85)
(John,CSE,72)
(Kiran,ECE,67)

Grouped Output

(CSE,{(Ravi,CSE,85),(John,CSE,72)})
(IT,{(Meera,IT,90)})
(ECE,{(Kiran,ECE,67)})

Joined Output

(Ravi,CSE,85,CSE,Dr.Sharma)
(John,CSE,72,CSE,Dr.Sharma)
(Kiran,ECE,67,ECE,Dr.Rao)
(Meera,IT,90,IT,Dr.Verma)

Result:

Thus, a Pig Latin script was successfully implemented to sort, group, join, project, and filter data, demonstrating Pig's ability to process structured datasets efficiently.

Ex No: 6 Implement Hive Databases, Tables, Views, Functions, and Indexes

AIM:

To create and demonstrate operations on Hive databases including table creation, views, indexes, and user-defined functions (UDFs) using a simulated Hive-like environment in Python.

Algorithm :

1. **Create Hive Database:**
   - Initialize a database (simulated here using SQLite for demonstration).
2. **Create Table:**
   - Define a table to store sample weather data with columns for `record_id`, `year`, and `temperature_c`.
3. **Insert Data:**
   - Load random weather data for multiple years into the table.
4. **Create Index:**
   - Create an index on the `year` column to optimize queries.
5. **Create View:**
   - Define a view (`positive_temps`) showing only records with temperature greater than 0°C.
6. **Create User-Defined Function (UDF):**
   - Implement a Python function to convert Celsius to Fahrenheit.
   - Register the UDF within the database.
7. **Query and Reporting:**
   - Generate reports of **min/max temperatures per year** using SQL queries.
   - Use the UDF within queries to convert values dynamically.

Python Implementation

```
import pandas as pd
import sqlite3
import random
from contextlib import contextmanager

# ==================================
# 2. Generate Sample Weather Data
# ==================================
def generate_sample_data(num_records=1000):
    years = list(range(1900, 2021))
    data = {
        'record_id': range(1, num_records + 1),
        'year': [random.choice(years) for _ in range(num_records)],
        'temperature_c': [random.uniform(-50, 50) for _ in range(num_records)]
    }
    return pd.DataFrame(data)
```

```
# ==================================
# 3. Simulate Hive Database & Table in SQLite
# ==================================
@contextmanager
def sqlite_connection(db_name):
    conn = sqlite3.connect(db_name)
    try:
        yield conn
    finally:
        conn.close()

def setup_hive_like_db():
    db_name = 'weather_hive.db'
    df = generate_sample_data(1000)

    with sqlite_connection(db_name) as conn:
        # Create Hive-like Table
        df.to_sql('weather_data', conn, if_exists='replace', index=False)

        # Create Index (simulating Hive CREATE INDEX)
        conn.execute('CREATE INDEX idx_year ON weather_data(year)')

        # Create View (simulating Hive CREATE VIEW)
        conn.execute('''
            CREATE VIEW positive_temps AS
            SELECT record_id, year, temperature_c
            FROM weather_data
            WHERE temperature_c > 0
        ''')

    print(f"Database '{db_name}', table 'weather_data', index 'idx_year', and view 'positive_temps' created
successfully.")

# ==================================
# 4. Create Hive-Like UDF (Function)
# ==================================
def celsius_to_fahrenheit(temp_c):
    return (temp_c * 9/5) + 32

def register_udf(conn):
    conn.create_function('c_to_f', 1, celsius_to_fahrenheit)
    print("User Defined Function (UDF) 'c_to_f' registered successfully.")

# ==================================
# 5. Generate Weather Report
# ==================================
def generate_weather_report():
    db_name = 'weather_hive.db'
    with sqlite_connection(db_name) as conn:
        register_udf(conn)

        # Query Table: Min/Max per Year
```

```
        query_table = '''
            SELECT year,
                MIN(temperature_c) AS min_temp_c,
                MAX(temperature_c) AS max_temp_c
            FROM weather_data
            GROUP BY year
            ORDER BY year
        '''
        report_df = pd.read_sql_query(query_table, conn)

        # Query View: Max Temp in Fahrenheit using UDF
        query_view = '''
            SELECT year,
                c_to_f(MAX(temperature_c)) AS max_temp_f
            FROM positive_temps
            GROUP BY year
            ORDER BY year
        '''
        view_df = pd.read_sql_query(query_view, conn)

        # Merge Both Results
        result = report_df.merge(view_df, on='year', how='left')
        result['max_temp_f'] = result['max_temp_f'].round(1)
        result['min_temp_c'] = result['min_temp_c'].round(1)
        result['max_temp_c'] = result['max_temp_c'].round(1)

    return result

# ================================
# 6. Main Execution
# ================================
if __name__ == "__main__":
    print("Setting up Hive-like environment...")
    setup_hive_like_db()

    print("\nGenerating Weather Temperature Statistics Report...")
    report = generate_weather_report()

    print("\n=== Weather Report ===")
    print("Year\tMin Temp (°C)\tMax Temp (°C)\tMax Temp (°F)")
    print("-" * 50)
    for _, row in report.iterrows():
        print(f"{int(row['year'])}\t{row['min_temp_c']}\t\t{row['max_temp_c']}\t\t{row['max_temp_f']}")

    print("\nSample Data from View (positive_temps):")
    with sqlite_connection('weather_hive.db') as conn:
        sample_view = pd.read_sql_query('SELECT * FROM positive_temps LIMIT 5', conn)
        print(sample_view)
```

Expected Output:

```
Setting up Hive-like environment...
Database 'weather_hive.db', table 'weather_data', index 'idx_year', and view 'positive_temps'
created successfully.

Generating Weather Temperature Statistics Report...
User Defined Function (UDF) 'c_to_f' registered successfully.

=== Weather Report ===
Year    Min Temp (°C)   Max Temp (°C)   Max Temp (°F)
-----------------------------------------
1900    -47.6           49.9            121.9
1901    -49.1           48.7            119.7
1902    -45.2           47.8            118.0
... (truncated) ...

Sample Data from View (positive_temps):
   record_id  year  temperature_c
0        2   1910         10.34
1       12   1954         24.76
2       25   1998          3.25
3       45   2009         47.92
4       52   1965         17.13
```

Result:

The Hive Experiment was successfully created using Python and SQLite to demonstrate database creation, tables, views, indexes, and user-defined functions. It efficiently generated analytical reports showing yearly temperature statistics in both Celsius and Fahrenheit.

Ex No: 7 Export Data from Hadoop using Sqoop and Import Data to Hive using Sqoop

AIM:

To simulate the process of exporting data from Hadoop Distributed File System (HDFS) and importing it into a Hive table using Sqoop, implemented using Python with SQLite and Pandas.

Algorithm :

1. Start the program.
2. Generate sample weather data (year-wise temperatures) and store it in a CSV file, simulating an HDFS file.
3. Establish a SQLite connection to simulate a Hive database.
4. Read the CSV file and import its data into the SQLite table, simulating the Sqoop import process.
5. Create an index on the year column to optimize query performance (like Hive index).
6. Query the table to calculate yearly minimum and maximum temperatures.
7. Display the summarized report and sample table data.
8. End the program.

Python Implementation

```
import pandas as pd
import sqlite3
import random
from contextlib import contextmanager

# Step 1: Generate sample weather data (simulating HDFS CSV file)
def generate_sample_data(num_records=1000):
    years = list(range(1900, 2021))
    data = {
        'record_id': range(1, num_records + 1),
        'year': [random.choice(years) for _ in range(num_records)],
        'temperature_c': [random.uniform(-50, 50) for _ in range(num_records)]
    }
    df = pd.DataFrame(data)
    csv_path = 'weather_data.csv'  # Simulating HDFS file
    df.to_csv(csv_path, index=False)
    print(f"Sample data generated and saved to {csv_path} (simulating HDFS file).")
    return csv_path

# Step 2: SQLite connection (simulating Hive)
@contextmanager
def sqlite_connection(db_name):
    conn = sqlite3.connect(db_name)
    try:
        yield conn
    finally:
        conn.close()
```

```
# Step 3: Simulate Sqoop export/import
def sqoop_like_import(csv_path, db_name, table_name):
    df = pd.read_csv(csv_path)
    print(f"Sqoop-like export: Read {len(df)} records from {csv_path} (HDFS).")
    with sqlite_connection(db_name) as conn:
        df.to_sql(table_name, conn, if_exists='replace', index=False)
        print(f"Sqoop-like import: Loaded data into {db_name}.{table_name} (Hive table).")
        conn.execute(f'CREATE INDEX idx_year ON {table_name}(year)')
        print(f"Index 'idx_year' created on {table_name}.year.")

# Step 4: Generate weather report
def generate_weather_report(db_name, table_name):
    with sqlite_connection(db_name) as conn:
        query = f"
            SELECT year,
                MIN(temperature_c) AS min_temp_c,
                MAX(temperature_c) AS max_temp_c
            FROM {table_name}
            GROUP BY year
            ORDER BY year
        "
        report_df = pd.read_sql_query(query, conn)
        report_df['min_temp_c'] = report_df['min_temp_c'].round(1)
        report_df['max_temp_c'] = report_df['max_temp_c'].round(1)
    return report_df

# Step 5: Run program
if __name__ == "__main__":
    print("=== Simulating Sqoop Export/Import to Hive ===")
    csv_path = generate_sample_data(1000)
    db_name = 'weather_hive.db'
    table_name = 'weather_data'
    sqoop_like_import(csv_path, db_name, table_name)

    print("\nGenerating Weather Temperature Statistics Report...")
    report = generate_weather_report(db_name, table_name)

    print("\n=== Weather Report ===")
    print("Year\tMin Temp (°C)\tMax Temp (°C)")
    print("-" * 35)
    for _, row in report.iterrows():
        print(f"{int(row['year'])}\t{row['min_temp_c']}\t\t{row['max_temp_c']}")

    print(f"\nSample data from {table_name} (first 5 rows):")
    with sqlite_connection(db_name) as conn:
        sample_data = pd.read_sql_query(f'SELECT * FROM {table_name} LIMIT 5', conn)
        print(sample_data)
```

Expected Output:

```
=== Simulating Sqoop Export/Import to Hive ===
Sample data generated and saved to weather_data.csv (simulating HDFS file).
Sqoop-like export: Read 1000 records from weather_data.csv (HDFS).
Sqoop-like import: Loaded data into weather_hive.db.weather_data (Hive table).
Index 'idx_year' created on weather_data.year.

Generating Weather Temperature Statistics Report...

=== Weather Report ===
Year    Min Temp (°C)   Max Temp (°C)
-----------------------------
1900    -48.7           49.2
1901    -44.3           47.9
1902    -46.1           48.5
...
2020    -49.6           49.9

Sample data from weather_data (first 5 rows):
   record_id  year  temperature_c
0          1  1915     -10.345678
1          2  1992      25.456789
2          3  2005      15.123456
3          4  1967     -22.987654
4          5  2018      40.678912
```

Result:

The simulation successfully demonstrated how Sqoop can export data from Hadoop (HDFS) and import it into Hive, using Python and SQLite as a lightweight prototype. It generated a summarized report of yearly minimum and maximum temperatures from the imported dataset.