**Exercise 2: Implementing Word Count using Hadoop MapReduce**

**Aim:**

To learn how to implement a Word Count program using Hadoop MapReduce in Python, demonstrating data processing using the mapper and reducer design patterns, while using Cloudera's environment.

**Procedure:**

* **Start Cloudera Services:**
  + Open Cloudera Manager and ensure that services like HDFS and YARN are running.
* **Create Three Files and Type the corresponding content**
  + **input.txt**
  + **mapper.py**
  + **reducer.py**
* Upload the input file to HDFS
  + hdfs dfs -mkdir -p /user/cloudera/wordcount/input
  + hdfs dfs -put input.txt /user/cloudera/wordcount/input/
* **Make the Mapper and Reducer Scripts Executable:**
  + chmod +x mapper.py
  + chmod +x reducer.py
* To run the mapper.py script directly on an input file like input.txt, you can use the command line.
  + cat input.txt | python mapper.py 🡺**Refer Output-1**
* To aggregate these counts, you'll need to pass this output to the reducer.
  + cat input.txt | python mapper.py | sort | python reducer.py **🡺Refer Output-2**

**Program:**

**input.txt**

Hadoop is great

Hadoop is scalable

Hadoop is open-source

**mapper.py**

#!/usr/bin/env python3

import sys

def mapper():

for line in sys.stdin:

line = line.strip() # Remove leading/trailing whitespace

words = line.split() # Split the line into words

for word in words:

print("%s\t%d" % (word, 1)) # Output word with a count of 1

if \_\_name\_\_ == "\_\_main\_\_":

mapper()

**reducer.py**

#!/usr/bin/env python3

import sys

def reducer():

current\_word = None

current\_count = 0

for line in sys.stdin:

line = line.strip()

word, count = line.split('\t')

try:

count = int(count)

except ValueError:

continue

if current\_word == word:

current\_count += count

else:

if current\_word:

print("%s\t%d" % (current\_word, current\_count))

current\_word = word

current\_count = count

if current\_word == word:

print("%s\t%d" % (current\_word, current\_count))

if \_\_name\_\_ == "\_\_main\_\_":

reducer()

**Output-1:** cat input.txt | python mapper.py

**Hadoop 1**

**is 1**

**great 1**

**Hadoop 1**

**is 1**

**scalable 1**

**Hadoop 1**

**is 1**

**open-source 1**

**Output-2:** cat input.txt | python mapper.py | sort | python reducer.py

**Hadoop 3**

**is 3**

**great 1**

**scalable 1**

**open-source 1**

### Result:

The Word Count program was successfully implemented using Hadoop MapReduce in Python on Cloudera. The program reads the input file, counts the occurrences of each word, and outputs the results using the format specifier method for stringrmatting.

**Exercise 3: Implementing Word Count by skip the stop words using Hadoop MapReduce**

**Aim:**

To implement a Word Count program using Hadoop MapReduce in Python that skips common stop words. This program will count the occurrences of each word from the input data while excluding words from a predefined list of stop words, demonstrating how to filter irrelevant terms from a dataset using the MapReduce paradigm.

**Procedure:**

* **Start Cloudera Services:**
  + Open Cloudera Manager and ensure that services like HDFS and YARN are running.
* **Create Three Files and Type the corresponding content**
  + **input.txt**
  + **mapper.py**
  + **reducer.py**
* Upload the input file to HDFS
  + hdfs dfs -mkdir -p /user/cloudera/wordcount/input
  + hdfs dfs -put input.txt /user/cloudera/wordcount/input/
* **Make the Mapper and Reducer Scripts Executable:**
  + chmod +x mapper.py
  + chmod +x reducer.py
* To run the mapper.py script directly on an input file like input.txt, you can use the command line.
  + cat input.txt | python mapper.py 🡺**Refer Output-1**
* To aggregate these counts, you'll need to pass this output to the reducer.
  + cat input.txt | python mapper.py | sort | python reducer.py **🡺Refer Output-2**

**Program:**

**input.txt**

Hadoop is great

Hadoop is scalable

Hadoop is open-source

**mapper.py**

#!/usr/bin/env python3

import sys

**# Define stop words as a list directly in the script**

stopwords = ["is", "a", "the", "for", "and", "of", "to", "in", "on", "with", "by", "it"]

def mapper():

for line in sys.stdin:

line = line.strip().lower() # Convert to lowercase for consistency

words = line.split() # Split the line into words

for word in words:

if word not in stopwords: # Skip the stop words

print("%s\t%d" % (word, 1)) # Output word with a count of 1

if \_\_name\_\_ == "\_\_main\_\_":

mapper()

**reducer.py**

#!/usr/bin/env python3

import sys

def reducer():

current\_word = None

current\_count = 0

for line in sys.stdin:

line = line.strip()

word, count = line.split('\t')

try:

count = int(count)

except ValueError:

continue

if current\_word == word:

current\_count += count

else:

if current\_word:

print("%s\t%d" % (current\_word, current\_count))

current\_word = word

current\_count = count

if current\_word == word:

print("%s\t%d" % (current\_word, current\_count))

if \_\_name\_\_ == "\_\_main\_\_":

reducer()

**Output-1:** cat input.txt | python mapper.py

**Hadoop 1**

**great 1**

**Hadoop 1**

**scalable 1**

**Hadoop 1**

**open-source 1**

**Output-2:** cat input.txt | python mapper.py | sort | python reducer.py

**Hadoop 3**

**great 1**

**scalable 1**

**open-source 1**

### Result:

The mapper will process the input, ignore the words in the stop words list, and count the occurrences of the remaining words. The final output will exclude common stop words.

### Exercise 3.1: Count the Occurrences of a List of Keywords from Given Text Files Using Hadoop MapReduce

#### Aim:

To count the occurrences of specific keywords (e.g., "Hadoop," "MapReduce," "data," "framework") from a text file using Hadoop MapReduce in Python. The goal is to track how many times the specified keywords appear in the input data, which can be useful for extracting insights from large datasets.

#### Procedure:

1. **Start Cloudera Services:**
   * Ensure that Cloudera Manager is running and services like HDFS and YARN are operational.
2. **Create Three Files with the Following Content:**
   * input-data.txt
   * mapper.py
   * reducer.py
3. **Upload the Input File to HDFS:**

bash

Copy code

hdfs dfs -mkdir -p /user/cloudera/keywordcount/input

hdfs dfs -put input-data.txt /user/cloudera/keywordcount/input/

1. **Make the Mapper and Reducer Scripts Executable:**

bash

Copy code

chmod +x mapper.py

chmod +x reducer.py

1. **Run the Mapper and Reducer Scripts:**

bash

Copy code

cat input-data.txt | python mapper.py | sort | python reducer.py

#### Program:

##### input-data.txt

kotlin

Copy code

Hadoop is an open-source framework for data storage and large-scale data processing.

It provides high availability and fault tolerance in distributed environments.

Organizations use Hadoop for handling massive amounts of structured and unstructured data.

##### mapper.py

python

Copy code

#!/usr/bin/env python3

import sys

# Define a list of keywords to track

keywords = ["hadoop", "mapreduce", "data", "framework"]

**def mapper():**

**for line in sys.stdin:**

**line = line.strip().lower() # Convert to lowercase for consistency**

**words = line.split() # Split the line into words**

**for word in words:**

**if word in keywords: # Check if the word is in the keywords list**

**print("%s\t%d" % (word, 1)) # Output keyword with a count of 1**

**if \_\_name\_\_ == "\_\_main\_\_":**

**mapper()**

##### reducer.py

#!/usr/bin/env python3

**import sys**

**def reducer():**

**current\_word = None**

**current\_count = 0**

**for line in sys.stdin:**

**line = line.strip()**

**word, count = line.split('\t')**

**try:**

**count = int(count)**

**except ValueError:**

**continue**

**if current\_word == word:**

**current\_count += count**

**else:**

**if current\_word:**

**print("%s\t%d" % (current\_word, current\_count))**

**current\_word = word**

**current\_count = count**

**if current\_word == word:**

**print("%s\t%d" % (current\_word, current\_count))**

**if \_\_name\_\_ == "\_\_main\_\_":**

**reducer()**

#### Output Example:

Assuming input-data.txt contains the text mentioned above:

**Mapper Output:**

kotlin

Copy code

hadoop 1

framework 1

data 1

hadoop 1

data 1

hadoop 1

**Reducer Output (after sorting):**

kotlin

Copy code

data 2

framework 1

hadoop 3

#### Result:

The mapper will process the input, emitting only the specified keywords along with a count of 1. The reducer will aggregate these counts, giving the total occurrences of each keyword. In the provided example, the keyword "Hadoop" appears 3 times, "data" appears 2 times, and "framework" appears 1 time.

4o

### Exercise 4: Loading and complex Data Transfromtions in Pig

### Aim

The goal is to load large datasets into Apache Pig, perform complex data transformations including filtering, joining, grouping, and advanced aggregations, and apply multiple filtering conditions.

### Procedure Breakdown

1. **Start Pig CLI or Grunt Shell**:
   * This is where you run the Pig scripts in your environment.
2. **Load the Datasets**:

pig

Copy code

-- Step 1: Load the employee dataset

employee\_data = LOAD 'employee\_data.csv' USING PigStorage(',')

AS (emp\_id:int, emp\_name:chararray, department\_id:int, salary:float, age:int);

-- Step 2: Load the department dataset

department\_data = LOAD 'department\_data.csv' USING PigStorage(',')

AS (department\_id:int, department\_name:chararray);

* + **LOAD**: Loads the specified CSV files into Pig.
  + **USING PigStorage(',')**: Specifies the delimiter used in the CSV file, which is a comma in this case.
  + **AS**: Defines the schema for the loaded data, specifying the data types for each field.

1. **Filter Employees**:

pig

Copy code

-- Step 3: Filter employees with salary greater than 70,000 and age greater than 30

filtered\_employees = FILTER employee\_data BY salary > 70000 AND age > 30;

* + **FILTER**: Filters the loaded employee data to include only those records where the salary is greater than 70,000 and age is greater than 30.

1. **Join Datasets**:

pig

Copy code

-- Step 4: Join the employee data with department data based on department\_id

joined\_data = JOIN filtered\_employees BY department\_id, department\_data BY department\_id;

* + **JOIN**: Combines the filtered employee data with the department data using the department\_id as the common key.

1. **Group Joined Data**:

pig

Copy code

-- Step 5: Group the joined data by department

grouped\_by\_department = GROUP joined\_data BY department\_data::department\_name;

* + **GROUP**: Groups the joined data by the department name, allowing for aggregation operations later.

1. **Calculate Aggregations**:

pig

Copy code

-- Step 6: Calculate total salary, average salary, and number of employees per department

department\_aggregates = FOREACH grouped\_by\_department GENERATE

group AS department\_name,

COUNT(joined\_data) AS employee\_count,

SUM(joined\_data.salary) AS total\_salary,

AVG(joined\_data.salary) AS avg\_salary;

* + **FOREACH ... GENERATE**: Iterates over the grouped data to calculate:
    - COUNT: The number of employees in each department.
    - SUM: The total salary of employees in each department.
    - AVG: The average salary of employees in each department.

1. **Filter Grouped Data**:

pig

Copy code

-- Step 7: Filter out departments with fewer than 2 employees

filtered\_departments = FILTER department\_aggregates BY employee\_count >= 2;

* + Filters the aggregated results to retain only those departments with at least 2 employees.

1. **Store Results**:

pig

Copy code

-- Step 8: Store the result in a new file filtered\_department\_summary

STORE filtered\_departments INTO 'filtered\_department\_summary' USING PigStorage(',');

* + **STORE**: Writes the final filtered results to a new file named filtered\_department\_summary, using a comma as the delimiter.

### Sample Datasets Explanation

1. **employee\_data.csv**:

python

Copy code

emp\_id,emp\_name,department\_id,salary,age

1,John Doe,101,85000,35

2,Jane Smith,102,72000,45

3,David Brown,101,95000,40

4,Mary Johnson,103,60000,29

5,Michael Lee,102,68000,32

...

1. **department\_data.csv**:

python

Copy code

department\_id,department\_name

101,Engineering

102,Marketing

103,Sales

104,HR

...

### Output Summary

* **Filtered Employees**: Shows employees with salaries greater than 70,000 and age greater than 30.
* **Aggregated Department Data**: Displays the total salary, average salary, and count of employees per department.
* **Filtered Department Data**: Presents the final output showing only departments that have at least 2 employees.

### Conclusion

This Pig script effectively loads datasets, performs a series of transformations, and generates meaningful insights through aggregations. It demonstrates the power of Apache Pig for handling complex data transformations with simple and readable syntax.

**Exercise 4.1 Loading and Complex Data Transformation in pig**

**Dataset Structure:**

1. **customers.csv**: { customer\_id, name, location }
2. **transactions.csv**: { transaction\_id, customer\_id, item, amount }

**Tasks Breakdown and Pig Script:**

1. **Load the datasets**:

pig

-- Load the customers data

customers = LOAD 'customers.csv' USING PigStorage(',')

AS (customer\_id:int, name:chararray, location:chararray);

-- Load the transactions data

transactions = LOAD 'transactions.csv' USING PigStorage(',')

AS (transaction\_id:int, customer\_id:int, item:chararray, amount:float);

1. **Filter Transactions**:
   * Filter transactions where the amount is greater than or equal to $1000.

pig

Copy code

-- Filter transactions with amount >= 1000

high\_value\_transactions = FILTER transactions BY amount >= 1000;

1. **Group Transactions by Customer**:
   * Group the filtered transactions by customer\_id.

pig

Copy code

-- Group transactions by customer\_id

grouped\_transactions = GROUP high\_value\_transactions BY customer\_id;

1. **Join Customer and Transaction Data**:
   * Join the customers and filtered transactions by customer\_id.

pig

Copy code

-- Join customer data with high value transactions based on customer\_id

customer\_transactions = JOIN customers BY customer\_id, high\_value\_transactions BY customer\_id;

1. **Calculate Total Spending per Customer**:
   * Calculate the total amount spent by each customer.

pig

Copy code

-- Calculate total spending per customer

customer\_spending = FOREACH grouped\_transactions GENERATE

group AS customer\_id,

SUM(high\_value\_transactions.amount) AS total\_spent;

1. **Find Top 3 Spenders**:
   * Sort the customers based on total amount spent and retrieve the top 3 customers.

pig

Copy code

-- Sort customers based on total spending and limit to top 3 spenders

top\_spenders = LIMIT (ORDER customer\_spending BY total\_spent DESC) 3;

1. **Group by Location and Calculate Total Spending per Location**:
   * Group customers by their location and calculate the total spending in each location.

pig

Copy code

-- Join customer details with total spending data

customer\_spending\_details = JOIN customers BY customer\_id, customer\_spending BY customer\_id;

-- Group by location and calculate total spending per location

location\_spending = GROUP customer\_spending\_details BY customers::location;

-- Calculate total spending per location

total\_spending\_per\_location = FOREACH location\_spending GENERATE

group AS location,

SUM(customer\_spending\_details::total\_spent) AS total\_spent;

**Output Explanation:**

1. **Filtered Transactions**:
   * Only transactions greater than or equal to $1000 are included.
2. **Grouped Transactions by Customer**:
   * For each customer, transactions are grouped to calculate total spending.
3. **Top 3 Spenders**:
   * The customers with the highest total spending are retrieved and displayed.
4. **Location-Based Spending**:
   * The total spending is calculated per customer location, showing high-value transactions across locations.

This Pig script demonstrates efficient data transformation techniques like filtering, grouping, joining, and aggregating data, allowing us to find high-value customers and analyze spending patterns across different locations.

**Exercise 1: Basic File and Directory Operations in Hadoop HDFS**

### Aim:

To perform basic file and directory operations in Hadoop HDFS, including creating directories, uploading files, listing directory contents, reading files, and deleting files and directories.

### Procedure:

* **Start Hadoop Services**:

Before performing any operations in HDFS, ensure that all the necessary Hadoop services are started, including the NameNode, DataNode, and ResourceManager.

* **Create a Directory in HDFS**:

Use the appropriate command to create a directory within the Hadoop Distributed File System (HDFS). This directory will be used to store files for further operations.

* **Upload Files to HDFS**:

Upload files from the local file system to the newly created directory in HDFS. This allows the files to be distributed across the HDFS storage nodes.

* **List Files and Directories in HDFS**:

List the files and subdirectories within the specified HDFS directory to verify the content.

* **View the Content of a File in HDFS**:

Display the content of a file stored in HDFS to ensure that the data is correctly uploaded.

* **Copy Files from HDFS to Local File System**:

Retrieve a file from HDFS back to the local file system. This is useful for downloading files after processing in Hadoop.

* **Delete Files from HDFS**:

Delete a specific file from the HDFS directory once it is no longer needed.

* **Delete Directories from HDFS**:

Recursively delete a directory and all its contents from HDFS.

* **Stop Hadoop Services** :

After performing the necessary file and directory operations, stop the Hadoop services to free up system resources. This step is optional depending on the environment.

**Program:**

**# Start Hadoop services**

**start-all.sh**

**# Create a new directory in HDFS**

**hdfs dfs -mkdir /user/cloudera/my\_directory**

**# Upload a file from local filesystem to HDFS**

**hdfs dfs -put /home/cloudera/sample.txt /user/cloudera/my\_directory/**

**# List files and directories in HDFS**

**hdfs dfs -ls /user/cloudera/my\_directory**

**# View the content of a file in HDFS**

**hdfs dfs -cat /user/cloudera/my\_directory/sample.txt**

**# Copy the file from HDFS to local filesystem**

**hdfs dfs -get /user/cloudera/my\_directory/sample.txt /home/cloudera/**

**# Delete a file from HDFS**

**hdfs dfs -rm /user/cloudera/my\_directory/sample.txt**

**# Delete the directory in HDFS recursively**

**hdfs dfs -rm -r /user/cloudera/my\_directory**

**# Stop Hadoop services (optional)**

**stop-all.sh**

**Result:**

Basic HDFS operations including creating directories, uploading files, listing, viewing, copying, and deleting files and directories were performed successfully.

### Exercise 1.1: Basic File and Directory Operations in Hadoop HDFS

#### Aim:

To perform basic file and directory operations in Hadoop HDFS, including creating directories, uploading files, listing contents, viewing files, copying files, renaming files, moving files, and deleting directories.

#### Procedure:

1. **Start Hadoop Services:** Before any HDFS operation, ensure all Hadoop services (NameNode, DataNode, ResourceManager, etc.) are running.

bash

Copy code

start-all.sh

1. **Create a Directory in HDFS:** Create a new directory called student\_data in your HDFS home directory.

bash

Copy code

hdfs dfs -mkdir /user/cloudera/student\_data

1. **Upload a File from Local Filesystem to HDFS:** Upload a file called records.txt from your local system to the student\_data directory in HDFS.

bash

Copy code

hdfs dfs -put /home/cloudera/records.txt /user/cloudera/student\_data/

1. **List All Files and Directories Inside student\_data:** Verify that the file records.txt has been successfully uploaded by listing the contents of the student\_data directory.

bash

Copy code

hdfs dfs -ls /user/cloudera/student\_data

1. **Display the Contents of records.txt in HDFS:** View the content of the records.txt file in HDFS to check if it was uploaded correctly.

bash

Copy code

hdfs dfs -cat /user/cloudera/student\_data/records.txt

1. **Copy the File records.txt from HDFS to Local Filesystem:** Copy records.txt from the HDFS student\_data directory to the local system’s Downloads folder.

bash

Copy code

hdfs dfs -get /user/cloudera/student\_data/records.txt /home/cloudera/Downloads/

1. **Rename the File records.txt to student\_records.txt:** Rename records.txt in the HDFS student\_data directory to student\_records.txt.

bash

Copy code

hdfs dfs -mv /user/cloudera/student\_data/records.txt /user/cloudera/student\_data/student\_records.txt

1. **Create a New Directory backup in HDFS and Move the File:** Create a new directory called backup in HDFS and move the renamed file student\_records.txt from student\_data to the backup directory.

bash

Copy code

hdfs dfs -mkdir /user/cloudera/backup

hdfs dfs -mv /user/cloudera/student\_data/student\_records.txt /user/cloudera/backup/

1. **Delete the student\_data Directory from HDFS:** Once the file has been moved, delete the student\_data directory from HDFS.

bash

Copy code

hdfs dfs -rm -r /user/cloudera/student\_data

1. **Remove the File student\_records.txt from the backup Directory:** After confirming the move, delete the student\_records.txt file from the backup directory in HDFS.

bash

Copy code

hdfs dfs -rm /user/cloudera/backup/student\_records.txt

1. **Delete the backup Directory from HDFS:** Finally, delete the backup directory from HDFS.

bash

Copy code

hdfs dfs -rm -r /user/cloudera/backup

1. **Stop Hadoop Services (Optional):** After performing the necessary operations, stop Hadoop services if needed to free up resources.

bash

Copy code

stop-all.sh

#### Result:

The HDFS operations, including creating directories, uploading, listing, displaying, copying, renaming, moving, and deleting files and directories, were successfully completed.