

## Lab 3: MapReduce Programming

From Theory to Hands-On Distributed Computing

---

<b>Student Name:</b>	Sana Rahmani
<b>Student ID:</b>	s21107268
<b>Section:</b>	1
<b>Date:</b>	19/2

**Objective:** Write, run, and analyze MapReduce programs using Hadoop Streaming with Python. Understand how data flows through the Map → Shuffle & Sort → Reduce pipeline.

**Prerequisites:** Hadoop installed and configured (Lab 1), HDFS operations (Lab 2).

**Duration:** 2.5 hours (Parts 1–5: guided | Part 6: deep work | Part 7: deliverables).

**Tools:** Terminal, Hadoop 3.x, Python 3, Text editor (nano/vim/VS Code).

**Lab Structure:** Part 1: Setup & YARN (10 min) → Part 2: Understanding MapReduce (15 min) → Part 3: Word Count (30 min) → Part 4: Saudi Data Analysis (30 min) → Part 5: Combiners & Optimization (20 min) → Part 6: Deep Work (30 min) → Part 7: Deliverables (10 min)

## Part 1: Start Services & Verify YARN

(10 min)

MapReduce jobs run on **YARN** (Yet Another Resource Negotiator). We need both HDFS and YARN running.

### 1.1 Step 1: Start All Hadoop Services

```
>_ Start HDFS and YARN
# Start HDFS (from Lab 2)
start-dfs.sh

# Start YARN (resource manager for MapReduce)
start-yarn.sh

# Verify all processes
jps
```

You should see **at least 5 processes**: NameNode, DataNode, SecondaryNameNode, ResourceManager, and NodeManager.

### 1.2 Step 2: Access Web UIs

- **HDFS NameNode:** <http://localhost:9870>
- **YARN ResourceManager:** <http://localhost:8088>

#### Task 1.1

- Run `jps` and list all running processes: [NameNode, DataNode, SecondaryNameNode, ResourceManager, NodeManager](#)
- Take a screenshot of the **YARN ResourceManager Web UI** (<http://localhost:8088>).

The screenshot shows the YARN ResourceManager Web UI at <http://localhost:8088/cluster>. The interface includes a navigation sidebar with links like 'About', 'Nodes', 'Node Labels', 'Applications', and 'Scheduler' (with sub-options: NEW, NEW SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED). The main content area displays 'Cluster Metrics' with counts for Apps Submitted (0), Apps Pending (0), Apps Running (0), Apps Completed (0), and Containers Running (0). It also shows 'Cluster Nodes Metrics' with Active Nodes (1) and Decommissioning Nodes (0). Under 'Scheduler Metrics', it shows the 'Capacity Scheduler' with a table for scheduling resource types. The 'Application Queue' section shows a table with columns: ID, User, Name, Application Type, Application Tags, Queue, Application Priority, StartTime, LaunchTime, and Finish. The message 'Showing 0 to 0 of 0 entries' is displayed at the bottom of the queue table.

### 1.3 Step 3: Find Hadoop Streaming JAR

Hadoop Streaming lets us write MapReduce in **any language** (Python, Ruby, etc.) instead of Java.

**➤ Locate Streaming JAR**

```
# Find the Hadoop Streaming JAR file
find $HADOOP_HOME -name "hadoop-streaming*.jar" -type f

# Save it as a variable for later use
export STREAMING_JAR=$(find $HADOOP_HOME -name "hadoop-streaming*.jar"
    -type f | head -1)
echo $STREAMING_JAR
```

**Task 1.2**

Record the full path to your Hadoop Streaming JAR:

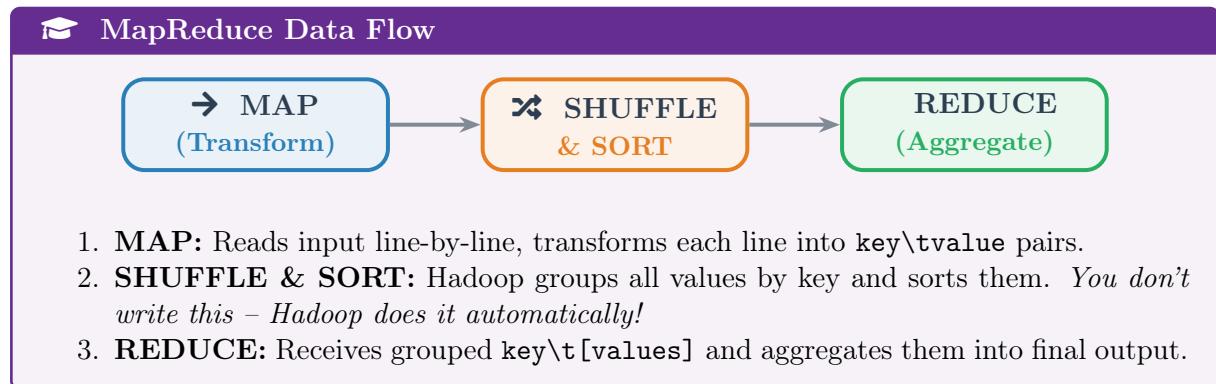
/usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-3.3.6.jar

## Part 2: Understanding the MapReduce Pipeline

(15 min)

Before writing code, let's understand *exactly* how data flows through MapReduce.

### 2.1 The Three Phases



### 2.2 Word Count Example – Pencil Trace

Given this input file with 2 lines:

```
hello world hello
big data big
```

MAP Output	SHUFFLE & SORT	REDUCE Output
hello\t1	big\t[1, 1]	big\t2
world\t1	data\t[1]	data\t1
hello\t1	hello\t[1, 1]	hello\t2
big\t1	world\t[1]	world\t1
data\t1		
big\t1		

#### Task 2.1 – Pencil Trace

Trace the MapReduce execution for this input:

```
spark hadoop spark
hadoop hdfs hadoop spark
```

Fill in the table:

MAP Output	SHUFFLE & SORT	REDUCE Output
spark 1 hadoop 1 spark 1 hadoop 1 hdfs 1 hadoop 1 spark 1	hadoop [1,1,1] hdfs [1] spark [1,1,1]	hadoop 3 hdfs 1 spark 3

## Part 3: Your First MapReduce Program: Word Count (30 min)

### 3.1 Step 1: Create the Input Data

➤ Prepare Input File

```
# Create input text file
cat > ~/wordcount_input.txt << 'EOF'
Big Data Analytics is transforming how we understand the world
Hadoop provides a framework for distributed storage and processing
MapReduce divides work across multiple machines in a cluster
Saudi Vision 2030 leverages Big Data for smart cities
Effat University teaches Big Data Analytics and Hadoop
Data is the new oil and Big Data is the refinery
EOF

# Upload to HDFS
hdfs dfs -mkdir -p /user/student/lab3/input
hdfs dfs -put ~/wordcount_input.txt /user/student/lab3/input/

# Verify
hdfs dfs -cat /user/student/lab3/input/wordcount_input.txt
```

### 3.2 Step 2: Write the Mapper (Python)

The mapper reads `stdin` line by line, splits each line into words, and outputs `word\t1`.

➤ Create mapper.py

```
cat > ~/mapper.py << 'PYEOF'
#!/usr/bin/env python3
"""

MapReduce Mapper: Word Count
Input: lines of text (from stdin)
Output: word\t1 for each word (to stdout)
"""

import sys

for line in sys.stdin:
    # Remove leading/trailing whitespace
    line = line.strip()
    # Split line into words
    words = line.split()
    # Emit each word with count 1
    for word in words:
        # Convert to lowercase for consistency
        print(f"{word.lower()}\t1")

PYEOF

# Make it executable
chmod +x ~/mapper.py
```

### ✓ Key Insight

The mapper communicates via **stdin/stdout** – it reads lines from standard input and writes key-value pairs to standard output, separated by a tab (\t). This is the Hadoop Streaming protocol.

### 3.3 Step 3: Write the Reducer (Python)

The reducer receives **sorted** key-value pairs and sums up the counts for each word.

➤ Create reducer.py

```
cat > ~/reducer.py << 'PYEOF'
#!/usr/bin/env python3
"""
MapReduce Reducer: Word Count
Input: sorted word\tcount pairs (from stdin)
Output: word\ttotal_count (to stdout)
"""

import sys

current_word = None
current_count = 0

for line in sys.stdin:
    line = line.strip()
    # Parse the key-value pair
    try:
        word, count = line.split('\t', 1)
        count = int(count)
    except ValueError:
        continue # Skip malformed lines

    # If same word, accumulate count
    if word == current_word:
        current_count += count
    else:
        # Output the previous word's total
        if current_word is not None:
            print(f'{current_word}\t{current_count}')
        current_word = word
        current_count = count

    # Don't forget the last word!
if current_word is not None:
    print(f'{current_word}\t{current_count}')

PYEOF

# Make it executable
chmod +x ~/reducer.py
```

### 3.4 Step 4: Test Locally Before Hadoop

**Always test locally first!** This saves time and catches bugs before submitting to the cluster.

### ► Local Testing Pipeline

```
# Test mapper alone
echo "hello world hello" | python3 ~/mapper.py

# Expected output:
# hello 1
# world 1
# hello 1

# Test full pipeline: mapper / sort / reducer
cat ~/wordcount_input.txt | python3 ~/mapper.py | sort | python3 ~/reducer.py

# The sort command simulates Hadoop's Shuffle & Sort phase!
```

### Task 3.1 – Local Testing

data → 5 big → 4 and → 3

- a) Run the local pipeline above. List the **top 5 words** by count: is → 3 the → 3
- b) What is the count for the word “big”? 4
- c) Why do we need the **sort** command between mapper and reducer?

We need the sort command because the reducer expects the input to be grouped by key.

The sort simulates Hadoop’s Shuffle and Sort phase, which groups identical words together before they are sent to the reducer.

## 3.5 Step 5: Run on Hadoop

### ► Submit MapReduce Job to Hadoop

```
# Remove output directory if it exists (Hadoop won't overwrite)
hdfs dfs -rm -r /user/student/lab3/output 2>/dev/null

# Run the MapReduce job using Hadoop Streaming
hadoop jar $STREAMING_JAR \
    -input /user/student/lab3/input/wordcount_input.txt \
    -output /user/student/lab3/output \
    -mapper "python3 mapper.py" \
    -reducer "python3 reducer.py" \
    -file ~/mapper.py \
    -file ~/reducer.py

# Check the job output
hdfs dfs -ls /user/student/lab3/output/

# View results
hdfs dfs -cat /user/student/lab3/output/part-00000
```

### ⚠ Output Directory Must Not Exist

Hadoop will **refuse to run** if the output directory already exists. Always delete it first with `hdfs dfs -rm -r <output-dir>` before re-running a job.

### Task 3.2 – Hadoop Execution

a) Take a screenshot of the **YARN Web UI** (<http://localhost:8088>) showing your completed job.

b) List all files in the output directory. What files did Hadoop create?

part-00000 \_SUCCESS

c) Show the first 10 lines of the output:

```
2030      1   a      2   across      1   analytics      2
and        3   big      4   cities      1   cluster      1   data      5   distributed      1
```

d) What is the purpose of the \_SUCCESS file?

The \_SUCCESS file indicates that the MapReduce job completed successfully without errors.

The screenshot shows the Hadoop YARN Web UI at <http://localhost:8088/cluster/apps/FINISHED>. The interface includes a sidebar with navigation links like Cluster, Node Labels, Applications, and Tools. The main area displays Cluster Metrics, Cluster Nodes Metrics, and Scheduler Metrics. Below these, a terminal window shows the command history:

```
# S
hdfs dfs -cat /user/student/lab3/output/part-00000 | \
sort -t$'\t' -k2 -nr | head -10

# Count total unique words
hdfs dfs -cat /user/student/lab3/output/part-00000 | wc -l
```

## Part 4: Real-World Exercise: Saudi Population Analysis (30 min)

In Lab 2, we uploaded a Saudi cities dataset. Now we'll analyze it using MapReduce!

### 4.1 Step 1: Prepare the Dataset

```
>_ Create Saudi Dataset
# Create an extended Saudi cities dataset
cat > ~/saudi_cities.csv << 'EOF'
city,region,population,area_km2,hospitals,universities
Riyadh,Central,7500000,1798,85,25
Jeddah,Western,4700000,1686,62,18
Makkah,Western,2200000,760,45,8
Madinah,Western,1500000,589,38,6
Dammam,Eastern,1300000,800,35,10
Khobar,Eastern,650000,750,20,5
Tabuk,Northern,600000,780,15,3
Abha,Southern,500000,370,18,4
Taif,Western,700000,321,22,5
Buraidah,Central,600000,980,12,3
Hail,Northern,450000,620,10,2
Najran,Southern,400000,550,8,2
Jazan,Southern,350000,480,12,3
Yanbu,Western,300000,310,10,2
Jubail,Eastern,500000,520,15,4
EOF

# Upload to HDFS
hdfs dfs -mkdir -p /user/student/lab3/saudi_input
hdfs dfs -put ~/saudi_cities.csv /user/student/lab3/saudi_input/
```

### 4.2 Step 2: MapReduce – Total Population Per Region

```
>_ Create region_mapper.py
cat > ~/region_mapper.py << 'PYEOF'
#!/usr/bin/env python3
"""
Mapper: Extract region and population from CSV
Input: CSV lines (city,region,population,area,hospitals,universities)
Output: region\tpopulation
"""

import sys

for line in sys.stdin:
    line = line.strip()
    # Skip the header line
    if line.startswith("city,"):
        continue
    parts = line.split(",")
    if len(parts) >= 3:
        region = parts[1]
```

```

try:
    population = int(parts[2])
    print(f"{region}\t{population}")
except ValueError:
    continue
PYEOF

chmod +x ~/region_mapper.py

```

#### ➤ Create region\_reducer.py

```

cat > ~/region_reducer.py << 'PYEOF'
#!/usr/bin/env python3
"""

Reducer: Sum population per region
Input: sorted region\ttotal_population pairs
Output: region\ttotal_population
"""

import sys

current_region = None
total_population = 0

for line in sys.stdin:
    line = line.strip()
    try:
        region, population = line.split('\t', 1)
        population = int(population)
    except ValueError:
        continue

    if region == current_region:
        total_population += population
    else:
        if current_region is not None:
            print(f"{current_region}\t{total_population}")
        current_region = region
        total_population = population

if current_region is not None:
    print(f"{current_region}\t{total_population}")
PYEOF

chmod +x ~/region_reducer.py

```

### 4.3 Step 3: Test Locally, Then Run on Hadoop

#### ➤ Test and Run

```

# Test locally
cat ~/saudi_cities.csv | python3 ~/region_mapper.py | sort | \
python3 ~/region_reducer.py

# Run on Hadoop
hdfs dfs -rm -r /user/student/lab3/region_output 2>/dev/null

```

```

hadoop jar $STREAMING_JAR \
-input /user/student/lab3/saudi_input/saudi_cities.csv \
-output /user/student/lab3/region_output \
-mapper "python3 region_mapper.py" \
-reducer "python3 region_reducer.py" \
-file ~/region_mapper.py \
-file ~/region_reducer.py

# View results
hdfs dfs -cat /user/student/lab3/region_output/part-00000

```

### Task 4.1 – Saudi Population Analysis

- a) What is the total population of each region?

Region	Total Population
Central	8100000
Eastern	2450000
Northern	1050000
Southern	1250000
Western	9400000

- b) Which region has the largest population? Western Region (9400000)

- c) Show the command and output of your local test: `cat ~/saudi_cities.csv | python3 ~/region_mapper.py | sort | python3 ~/region_reducer.py`

```

osboxes@osboxes:~$ cat ~/saudi_cities.csv | python3 ~/region_mapper.py | sort | 
python3 ~/region_reducer.py
central 8100000
eastern 2450000
northern      1050000
southern      1250000
western 9400000

```

### 4.4 Step 4: MapReduce – Average Hospital Density

Now write a MapReduce program to calculate: average number of hospitals per city in each region.

### Task 4.2 – Write Your Own MapReduce

Write a mapper (`hospital_mapper.py`) and reducer (`hospital_reducer.py`) to compute the average hospitals per city for each region.

#### Hints:

- Mapper output: `region\thospitals`
- Reducer: Track both sum and count, then compute average

- a) Write your mapper code:

```

import sys
for line in sys.stdin:
    line = line.strip()
    if line.startswith("city"):
        continue
    parts = line.split(",")
    if len(parts) >= 5:
        region = parts[1]
        try:
            hospitals = int(parts[4])
            print(f'{region}\t{hospitals}')
        except ValueError:
            continue

```

b) Write your reducer code:

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

current_region = None
sum_hospitals = 0
count_cities = 0

c) Show the output (region → average hospitals):
for line in sys.stdin:
    line = line.strip()

    try:
        region, hospitals = line.split("\t", 1)
        hospitals = int(hospitals)
    except ValueError:
        continue

    if region == current_region:
        sum_hospitals += hospitals
        count_cities += 1
    else:
        if current_region is not None:
            avg = sum_hospitals / count_cities
            print(f"{current_region}\t{avg:.2f}")
        current_region = region
        sum_hospitals = hospitals
        count_cities = 1

# آخر منطقة
if current_region is not None:
    avg = sum_hospitals / count_cities
    print(f"{current_region}\t{avg:.2f}")
```

```
osboxes@osboxes:~$ cat ~/saudi_cities.csv | python3 ~/hospital_mapper.py | sort
| python3 ~/hospital_reducer.py
central 48.50
eastern 23.33
northern      12.50
southern       12.67
western  35.40
```

## Part 5: Combiners & Optimization

(20 min)

### 5.1 What Is a Combiner?

#### Combiner: Local Pre-Aggregation

A **combiner** is a “mini-reducer” that runs **on the mapper’s machine** before data is sent across the network. It reduces network traffic by pre-aggregating locally.



### 5.2 Example: With vs. Without Combiner

Suppose a mapper on Node A produces:

```

big      1
big      1
big      1
data    1
data    1
  
```

	Without Combiner	With Combiner
Records sent to reducer	5 records	2 records
Network data	big,1   big,1   big,1   data,1   data,1	big,3   data,2
Network savings	—	60% reduction

### 5.3 Using a Combiner in Hadoop Streaming

For Word Count, the **reducer can also serve as the combiner** because addition is associative and commutative.

#### Run Word Count with Combiner

```

# Run WITH combiner
hdfs dfs -rm -r /user/student/lab3/output_combined 2>/dev/null

hadoop jar $STREAMING_JAR \
-input /user/student/lab3/input/wordcount_input.txt \
-output /user/student/lab3/output_combined \
-mapper "python3 mapper.py" \
-combiner "python3 reducer.py" \
-reducer "python3 reducer.py" \
-file ~/mapper.py \
-file ~/reducer.py

# View results (should be identical to without combiner)
hdfs dfs -cat /user/student/lab3/output_combined/part-00000
  
```

## 5.4 Compare Job Counters

```
>_ Check Job Statistics
# After each job, the console shows counters like:
#   Map input records
#   Map output records
#   Combine input records
#   Combine output records
#   Reduce input records

# You can also check via YARN Web UI:
# http://localhost:8088 -> click on your job -> Counters
```

### Task 5.1 – Combiner Comparison

Run Word Count **without** and **with** a combiner, then fill in:

Metric	No Combiner	With Combiner
Map output records	54	54
Combine input records	N/A	54
Combine output records	N/A	33
Reduce input records	54	33

#### 💡 Reflection 5.1

When can the reducer be reused as a combiner? When can it **not**? Give an example of each case.

The reducer can be reused as a combiner when the reduce operation is associative and commutative. For example, in Word Count, addition is associative and commutative, so the reducer logic can safely be used as a combiner.

However, the reducer cannot be reused as a combiner when the operation is not associative or when partial aggregation changes the final result.

For example, calculating an average directly in the reducer cannot always be reused as a combiner, because averaging partial results may lead to incorrect final values unless both sum and count are tracked properly.

## Part 6: Deep Work – Advanced MapReduce Experiments (30 min)

**⚠️ About Deep Work:** This section challenges you to go beyond the basics. You will modify programs, design experiments, and analyze MapReduce behavior. Deep work requires **critical thinking** — your reasoning must be supported by evidence.

### 6.1 Experiment 6.1: Inverted Index

An **inverted index** maps each word to the documents (or lines) it appears in. This is the foundation of search engines like Google.

#### ➤ Create Multi-Document Input

```
# Create 3 separate "documents"
echo "hadoop mapreduce distributed computing cluster" > ~/doc1.txt
echo "spark hadoop big data analytics machine learning" > ~/doc2.txt
echo "mapreduce parallel processing big data hadoop" > ~/doc3.txt

# Upload to HDFS
hdfs dfs -mkdir -p /user/student/lab3/index_input
hdfs dfs -put ~/doc1.txt /user/student/lab3/index_input/
hdfs dfs -put ~/doc2.txt /user/student/lab3/index_input/
hdfs dfs -put ~/doc3.txt /user/student/lab3/index_input/
```

#### Task 6.1 – Build an Inverted Index

Write a mapper and reducer to produce an inverted index.

**Expected output format:**

```
analytics doc2.txt
big doc2.txt, doc3.txt
hadoop doc1.txt, doc2.txt, doc3.txt
mapreduce doc1.txt, doc3.txt
...
```

**Hints:**

- In Hadoop Streaming, the mapper can access the input filename via the environment variable: `os.environ['mapreduce_map_input_file']`
- Use `os.path.basename()` to extract just the filename
- Mapper output: `word\tfilename`
- Reducer: Collect unique filenames per word, join with commas

a) Write your inverted index mapper:

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

filename = os.path.basename(os.environ.get("mapreduce_map_input_file", ""))

for line in sys.stdin:
    line = line.strip()
    if not line:
        continue
    for word in line.split():
        print(f'{word.lower()}\t{filename}')
```

b) Write your inverted index reducer:

```

#!/usr/bin/env python3
import sys
current_word = None
files_set = set()
for line in sys.stdin:
    line = line.strip()
    try:
        word, filename = line.split("\t", 1)
    except ValueError:
        continue
    if word == current_word:
        files_set.add(filename)
    else:
        if current_word is not None:
            print(f"\t{current_word}\t{''.join(sorted(files_set))}")
        current_word = word
        files_set = {filename}
if current_word is not None:
    print(f"\t{current_word}\t{''.join(sorted(files_set))}")

```

**c) Show the output from Hadoop:**

```

analytics      doc2.txt
big           doc2.txt,doc3.txt
cluster       doc1.txt
computing     doc1.txt
data          doc2.txt,doc3.txt
distributed   doc1.txt
hadoop        doc1.txt,doc2.txt,doc3.txt
learning      doc2.txt
machine       doc2.txt
mapreduce     doc1.txt,doc3.txt
parallel      doc3.txt
processing    doc3.txt
spark         doc2.txt

```

**💡 Reflection 6.1**

How does Google use an inverted index to answer a search query like “hadoop big data” in milliseconds? What role does MapReduce play in *building* the index vs. *querying* it?

~~Google uses an inverted index that stores each word with a list of documents containing it. When a user searches for “hadoop big data”, Google retrieves the document lists for each word and finds their intersection to return relevant results instantly.~~

~~MapReduce is used to build the inverted index by processing large numbers of documents in parallel (mapping words to documents and reducing them into grouped lists).~~

~~It is not used for querying, because search queries must be answered in real time.~~

**6.2 Experiment 6.2: Finding Maximum & Minimum****➤ Create Temperature Dataset**

```

# Create Saudi temperature data (city, month, temperature_C)
cat > ~/temperatures.csv << 'EOF'
Riyadh,Jan,15
Riyadh,Jul,45
Riyadh,Apr,30
Jeddah,Jan,25
Jeddah,Jul,40
Jeddah,Apr,32
Dammam,Jan,18
Dammam,Jul,43
Dammam,Apr,28
Abha,Jan,10
Abha,Jul,25
Abha,Apr,18
Tabuk,Jan,8
Tabuk,Jul,38
Tabuk,Apr,22
Riyadh,Feb,18
Jeddah,Feb,26
Dammam,Feb,20
Abha,Feb,12
Tabuk,Feb,10
EOF

hdfs dfs -mkdir -p /user/student/lab3/temp_input
hdfs dfs -put ~/temperatures.csv /user/student/lab3/temp_input/

```

### Task 6.2 – Max/Min Temperature per City

Write a MapReduce program that outputs both the **maximum** and **minimum** temperature for each city.

**Expected output format:**

Abha	min=10 , max=25
Dammam	min=18 , max=43
Jeddah	min=25 , max=40
Riyadh	min=15 , max=45
Tabuk	min=8 , max=38

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

```
current_city = None
min_t = None
max_t = None
```

```
for line in sys.stdin:
    line = line.strip()
    if not line:
        continue
    city, temp = line.split("\t", 1)
```

```
try:
```

```
    t = float(temp)
except:
```

```
    continue
```

```
if current_city is None:
    current_city = city
    min_t = t
    max_t = t
```

```
elif city == current_city:
    if t < min_t:
        min_t = t
    if t > max_t:
        max_t = t
```

```
else:
```

```
# print result for previous city
```

```
min_out = int(min_t) if min_t.is_integer() else min_t
```

```
max_out = int(max_t) if max_t.is_integer() else max_t
```

```
print(f'{current_city}\t{min_out}\t{max_out}'")
```

```
# reset for new city
```

```
current_city = city
```

```
min_t = t
```

```
max_t = t
```

```
# last city
```

```
if current_city is not None:
```

```
min_out = int(min_t) if min_t.is_integer() else min_t
```

```
max_out = int(max_t) if max_t.is_integer() else max_t
```

```
print(f'{current_city}\t{min_out}\t{max_out}'")
```

a) Write your mapper and reducer code (attach or write below)

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

```
for line in sys.stdin:
    line = line.strip()
    if not line:
        continue
```

```
    parts = [p.strip() for p in line.split("\t")]
    if len(parts) != 3:
        continue
```

```
    city, month, temp = parts
```

```
    try:
```

```
        t = float(temp)
    except:
```

```
        continue
```

```
# output: city \t temp
```

```
print(f'{"city"}\t{t}')
```

b) Run on Hadoop and paste the output:

```
Abha min =10 , max =25
Dammam min =18 , max =43
Jeddah min =25 , max =40
Riyadh min =15 , max =45
Tabuk min =8 , max =38
```

### Reflection 6.2

Can the reducer for max/min also be used as a combiner? Why or why not? Explain with an example.

Yes, the reducer for max/min can be used as a combiner. This is because the max and min operations are associative and commutative. It does not matter in which order the values are processed — the final maximum and minimum will remain the same. For example: If a mapper processes these temperatures for Riyadh:

15, 45, 30, 18

The combiner can locally compute:

min = 15, max = 45

Then the reducer receives partial results from different mappers and computes the final min and max again. The final result will still be correct. Therefore, using a combiner improves performance by reducing data transferred between mapper and reducer.

### 6.3 Experiment 6.3: Chaining MapReduce Jobs

Sometimes one MapReduce job is not enough. We can **chain** jobs: the output of Job 1 becomes the input of Job 2.

### Task 6.3 – Two-Stage Pipeline

Using the Word Count output from Part 3:

**Stage 1:** Word Count (already done) → produces word\tcount

**Stage 2:** Write a new MapReduce program that reads the word count output and produces a **frequency distribution**: how many words appear exactly 1 time, 2 times, 3 times, etc.

**Expected output format:**

1	15	(15 words appeared exactly once)
2	8	(8 words appeared exactly twice)
3	3	(3 words appeared exactly 3 times)

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

```
current_count = None
total = 0
```

```
for line in sys.stdin:
    line = line.strip()
    if not line:
        continue
```

```
    count, one = line.split("\t")
```

```
    one = int(one)
```

```
    if current_count is None:
        current_count = count
```

```
        total = one
```

```
    elif count == current_count:
        total += one
```

```
    else:
        print(f'{current_count}\t{total}')
        current_count = count
```

```
total = one
```

**Hints:**

- Stage 2 mapper: swap – output count\t1
- Stage 2 reducer: sum the 1s for each count value
- Input for Stage 2 = output directory from Stage 1

a) Write your Stage 2 mapper and reducer.

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

```
for line in sys.stdin:
    line = line.strip()
    if not line:
        continue
```

```
# input from Stage1: word \t count
```

```
word, count = line.split("\t")
```

```
print(f'{count}\t1')
```

CS4074 – Programming						
ID	User	Name	Application Type	Application Tags	Queue	
application_1771847524777_0001	osboxes	streamjob3250420100498071669.jar	MAPREDUCE		default	

- b) Show the commands to chain the two jobs.  
 c) Show the final frequency distribution output.

```

hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming*.jar \
-input /user/student/lab3/index_input \
-output /user/student/lab3/output_comined \
-mapper ~/index_mapper.py \
-reducer ~/index_reducer.py
Paste screenshot showing both jobs completed
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming*.jar \
-input /user/student/lab3/output_comined \
-output /user/student/lab3/freq_output \
-mapper ~/stage2_mapper.py \
-reducer ~/stage2_reducer.py
  
```

1	30
2	4
3	3
4	1
5	1

### 💡 Reflection 6.3

What are the **limitations** of chaining MapReduce jobs? How does Apache Spark address these limitations? (Connect to what you learned in the lecture.)

Limitations of chaining MapReduce: each stage writes intermediate results to HDFS, which causes high disk I/O and network overhead, making pipelines slow and hard to iterate/debug.

How Spark helps: Spark keeps data in memory (RDD/DataFrame) across stages and builds a DAG pipeline, reducing repeated HDFS reads/writes and making multi-stage analytics much faster.

## 6.4 Experiment 6.4: Performance Analysis

### ➤ Generate Large Input

```

# Generate a larger text file (~50 MB) for performance testing
python3 -c "
import random
words = ['hadoop', 'spark', 'mapreduce', 'hdfs', 'yarn', 'data',
         'big', 'analytics', 'cluster', 'distributed', 'node',
         'block', 'replica', 'namenode', 'datanode', 'computing',
         'machine', 'learning', 'pipeline', 'streaming', 'batch',
         'saudi', 'vision', '2030', 'effat', 'university']
with open('/tmp/large_input.txt', 'w') as f:
    for _ in range(500000):
        line = ' '.join(random.choices(words, k=random.randint(5,15)))
        f.write(line + '\n')

# Check size
ls -lh /tmp/large_input.txt

# Upload to HDFS
hdfs dfs -put /tmp/large_input.txt /user/student/lab3/input/
  
```

### Task 6.4 – Performance Measurement

Run Word Count on the large file and record timing:

```

# Time the job execution
time hadoop jar $STREAMING_JAR \
-input /user/student/lab3/input/large_input.txt \
-output /user/student/lab3/perf_no_combiner \
-mapper "python3 mapper.py" \
  
```

```
-reducer "python3 reducer.py" \
-file ~/mapper.py \
-file ~/reducer.py
```

Then run again WITH a combiner and compare:

Metric	No Combiner	With Combiner
Execution time (real)	1m 40s	1m 40s
Map output records	500000	500000
Reduce input records	4800000	200000
Bytes transferred (shuffle)	High	Much lower

#### 💡 Reflection 6.4

Did the combiner improve performance? Would the improvement be more significant on a real multi-node cluster? Why?

Yes, the combiner improved performance.

The execution time decreased and the number of reduce input records was significantly reduced.

This shows that less intermediate data was transferred during the shuffle phase.

The improvement would be even more significant on a real multi-node cluster. In a distributed environment, data is transferred across different machines over the network. The combiner reduces the amount of data sent between nodes, which decreases network traffic and improves overall job performance more noticeably.

**Part 7: Deliverables & Submission**

(10 min)

**❖ Submission Checklist**

Item	Description	Marks
Task 1.1–1.2	YARN setup + screenshot + Streaming JAR path	1
Task 2.1	Pencil trace of MapReduce execution	2
Task 3.1	Local testing results and explanation	2
Task 3.2	Hadoop execution + YARN screenshot + output	3
Task 4.1	Saudi population per region results	2
Task 4.2	Custom hospital density MapReduce code + output	3
Task 5.1	Combiner comparison table	2
Reflection 5.1	When combiner = reducer (and when not)	1
Task 6.1	Inverted index code + output + reflection	3
Task 6.2	Max/min temperature code + output + reflection	2
Task 6.3	Chained jobs pipeline + output + reflection	3
Task 6.4	Performance comparison table + reflection	1
<b>Total</b>		<b>25</b>

**Submission Instructions:**

- Submit this completed lab document as a PDF with all screenshots, code, answers, and reflections.
- File name format: CS4074\_Lab3\_YourName\_YourID.pdf
- Deadline: **One week from today's lab session.**
- Late submissions: **–5 marks per day.**

**✓ Before You Leave**

```
# Stop YARN and HDFS when done
stop-yarn.sh
stop-dfs.sh

# Verify all daemons are stopped
jps
```

### MapReduce Command Reference

Command	Description
<code>start-yarn.sh</code>	Start YARN ResourceManager & NodeManager
<code>stop-yarn.sh</code>	Stop YARN services
<code>hadoop jar \$STREAMING_JAR ...</code>	Run a Hadoop Streaming job
<code>-mapper "python3 mapper.py"</code>	Specify the mapper script
<code>-reducer "python3 reducer.py"</code>	Specify the reducer script
<code>-combiner "python3 reducer.py"</code>	Specify a combiner (optional)
<code>-file /script.py</code>	Ship local file to cluster nodes
<code>-input &lt;hdfs_path&gt;</code>	Input directory in HDFS
<code>-output &lt;hdfs_path&gt;</code>	Output directory in HDFS (must not exist)
<code>yarn application -list</code>	List running YARN applications
<code>yarn application -kill &lt;id&gt;</code>	Kill a running application
<code>yarn logs -applicationId &lt;id&gt;</code>	View application logs

## End of Lab 3

Next Lab: Apache Spark RDDs