# Lab 5: Running Hadoop Job on NYU HPC

# Lab 5 outline

1. Create GitHub repo & upload dataset
2. Connect to NYU Dataproc cluster
3. Linux warm-up commands
4. Write the MapReduce job (Python)
5. Run locally to debug
6. Run on Hadoop cluster
7. Retrieve results from HDFS

# Why This Lab?

### HW3 Preparation

Homework 3 requires running real Hadoop jobs on this exact cluster. This lab is your practice run so nothing is new on submission day.

### Hands-On Skills

You learn the complete workflow: write code → test locally → deploy on cluster → retrieve results from HDFS.

### Linux commands

This mirrors how data engineers work with the linux commands

# The Big Data Problem

### One Laptop

Limited to 4–16 CPU cores

RAM maxes out at 8–32 GB

Disk I/O becomes bottleneck

Crashes if data is too large

### Real Companies Use Clusters

**Billions of logs** — per second — clicks, errors, events

**Billions of purchases** — e-commerce transactions worldwide

**Petabytes of data** — 1 PB = 1,000 TB = 1,000,000 GB

**Hundreds of machines** — working together as ONE system

# What is HPC?

HPC = High Performance Computing — many machines working as one
Discussion from lecture

### Many machines, one system

Dozens to thousands of servers networked together. Your job runs across all of them simultaneously — like having thousands of CPUs.

### Shared resources

CPUs, memory, and storage are pooled and shared. You request resources, the cluster allocates them, and releases them when your job finishes.

### NYU Dataproc cluster

NYU provides a Google Dataproc cluster — a fully managed Hadoop/Spark environment. You connect to it via a web terminal and run jobs just like on a local machine.

# What is Hadoop?

Open-source framework for storing and processing large datasets across many machines

## 1 HDFS

Hadoop Distributed File System

- Splits large files into 128 MB blocks
- Each block stored on multiple machines (replication = safety)
- If one machine fails, data is safe on others
- Think of it as a distributed hard drive

## 2 MapReduce

Distributed Computation Engine

- Breaks computation into parallel tasks
- Sends each task to the machine holding the data
- "Compute where data lives" — avoids network transfer
- Results automatically merged (reduced) at the end

# What is MapReduce?

"Distributed GROUP BY" — think of it as SQL's GROUP BY running across hundreds of machines in parallel

**① MAP**

**Reads each row, emits a (key, value) pair**

`Example:  Row: "Alice,2024,Milk,Grocery,3.50"  →  emit("Grocery", 1)`

💡 *Converts raw rows into structured key-value pairs Hadoop can group*

**② SHUFFLE & SORT**

**Hadoop groups all values with the same key — you never write this code**

`Example:  "Grocery" → [1, 1, 1, 1, 1, ...]`

💡 *Happens automatically between Map and Reduce — Hadoop does it behind the scenes*

**③ REDUCE**

**Receives one key + all its values, outputs the final aggregated result**

`Example:  "Grocery", [1,1,1,…]  →  emit("Grocery", 40)`

💡 *Produces your answer — one output row per unique key*

# Our Dataset

shopping_data_200.csv — 200 rows of shopping transactions

| Column | Type | Meaning | Example |
|---|---|---|---|
| `user_id` | TEXT | Unique customer identifier | e.g., U001, U042 |
| `date` | DATE | Transaction date | e.g., 2024-01-15 |
| `item` | TEXT | Product purchased | e.g., Milk, Laptop, Pen |
| **`category`** ⭐ | TEXT | Our GROUP BY key (column index 3) | Grocery, Electronics… |
| `price` | FLOAT | Purchase amount in USD | e.g., 3.50, 299.99 |

```
Sample:   U012,2024-01-15,Milk,Grocery,3.50
```

**Goal: Count purchases per category → MapReduce GROUP BY**

# SQL vs MapReduce — Same Goal, Different Scale

## SQL Approach

```
SELECT category, COUNT(*)
FROM shopping_data
GROUP BY category;
```

Easy to write & read

Great for moderate data

Runs on ONE machine only

Fails on very large datasets

## MapReduce Approach

```
mapper:   row  →  emit(category, 1)
shuffle:  Hadoop groups all same keys
(automatic)
reducer:  (category, [1,1,1,…])  →  sum  →
output
```

Runs across many machines

Scales to petabytes

More code to write

Higher setup overhead

# Step 1 — Create Your GitHub Repo

① **Create a new repository**

On github.com → New → Name it exactly: lab5-hpc-mapreduce (exact name used for grading)

② **Create two folders**

data/ → will contain your CSV dataset
src/ → will contain your Python MapReduce code

③ **Upload the dataset**

Drag data/shopping_data_200.csv into the data/ folder via GitHub web UI, then commit

④ **Verify repo structure**

```
lab5-hpc-mapreduce/
 ├── data/shopping_data_200.csv
 └── src/  (empty for now — code added later)
```

# Step 2 — Connect to NYU Dataproc

Open the Dataproc Web Terminal provided by your TA, then run:

```
$ mkdir lab5
```

mkdir = make directory. Creates a new folder named "lab5" in your home directory

*Keeps all your lab files organized in one place — not scattered around*

```
$ cd lab5
```

cd = change directory. Moves your terminal prompt into the lab5 folder

*Like double-clicking a folder on Windows/Mac. All commands now run inside this folder*

```
$ git clone https://github.com/<you>/lab5-hpc-mapreduce.git
```

git clone = downloads an entire GitHub repo to the cluster machine

*Copies your code and data from GitHub onto Dataproc so Hadoop can access them*

```
$ cd lab5-hpc-mapreduce
```

Moves into the cloned repo folder

*You must be inside the repo directory for all remaining commands to work correctly*

# Step 3 — Linux Warm-Up Commands

Run these to verify your files are in place and explore the dataset

| | |
| --- | --- |
| `$ ls` | list → shows all files and folders in current directory<br>→ `data/    src/` |
| `$ ls data` | list inside data/ → shows what files are in that folder<br>→ `shopping_data_200.csv` |
| `$ pwd` | print working directory → shows your exact current path on the cluster<br>→ `/home/user/lab5/lab5-hpc-mapreduce` |
| `$ head data/shopping_data_200.csv` | show first 10 lines of the file — quick sanity check to confirm columns<br>→ `user_id,date,item,category,price`<br>`U001,2024-01-01,Milk,Grocery,3.50 ...` |
| `$ wc -l data/shopping_data_200.csv` | word count -lines → counts rows. 201 = 200 data rows + 1 header row<br>→ `201 data/shopping_data_200.csv` |
| `$ head -n 1 data/... \| tr ',' '\n' \| nl` | pipe: 1st row → replace commas with newlines → number each line = find column index<br>→ `1 user_id   2 date   3 item   4 category   5 price` |

# Step 3b — Search Data with grep

grep = global regular expression print — filters lines containing a pattern

```
$ grep Grocery data/shopping_data_200.csv | head
```

### `grep Grocery`

**grep = filter lines**

Reads every line and keeps ONLY lines that contain the word "Grocery". All other lines are discarded. Works like Ctrl+F on the entire file.

### `| (pipe)`

**| = pipeline connector**

The pipe | takes the OUTPUT of the command on the LEFT and feeds it as INPUT to the command on the RIGHT. Data flows through like water in a pipe.

### `head`

**head = show first 10**

Without head, ALL matching rows would print — potentially thousands. head limits output to just 10 rows so you can quickly check the results.

```
Sample output:  U001,2024-01-01,Milk,Grocery,3.50    U007,2024-01-08,Bread,Grocery,2.20    ...
```

# Step 4 — Create the MapReduce File

```
$ nano src/mr_sales_per_category.py
```

nano is a command-line text editor built into Linux. It opens a file for editing directly inside the terminal — no GUI, no mouse needed.
After pasting your code: Ctrl+X → press Y → press Enter to save and exit.

**Understanding the file path:**

| src/ | mr_ | sales_per_category | .py |
|---|---|---|---|
| Folder — keeps code separate from data | "mr_" prefix = MapReduce (naming convention) | Describes what the job computes | Python file extension |

After saving → commit & push to GitHub: git add src/mr_sales_per_category.py && git commit -m "add MR job" && git push

# Step 4b — The MapReduce Code (Full Walkthrough)

```
from mrjob.job import MRJob
import csv

class MRSalesPerCategory(MRJob):
  def mapper(self, _, line):
    if "user_id" in line:
      return
    row = next(csv.reader([line]))
    category = row[3]
    yield category, 1

  def reducer(self, key, values):
    yield key, sum(values)

if __name__ == "__main__":
  MRSalesPerCategory.run()
```

① **Import MRJob framework**

mrjob = Python library that simplifies Hadoop MapReduce. import csv = handles CSV line parsing.

② **Skip header row**

"user_id" in line detects the header. return exits mapper early so we don't count column names.

③ **Parse CSV & emit (key, 1)**

csv.reader parses the line. row[3] = 4th column = category. yield sends (category, 1) to shuffle.

④ **Reducer sums all 1s**

Hadoop calls reducer once per unique category with all emitted 1s. sum(values) = total purchase count.

⑤ **Entry point**

.run() tells MRJob to start the MapReduce job when this script is executed directly.

# Step 5 — Run Locally (Debug Mode)

Always test locally before submitting to the cluster — cluster errors are much harder to diagnose

```
$ python3 src/mr_sales_per_category.py data/shopping_data_200.csv
```

| `python3` | `src/mr_sales_per_category.py` | `data/shopping_data_200.csv` |
|---|---|---|
| Run with Python 3 interpreter (not python2 — important!) | Your MapReduce script relative to current directory | Input file — no cluster flag means LOCAL mode |

**What happens in local mode:**

① MRJob calls mapper() on each row of your CSV — simulating the Map phase

② MRJob groups and sorts output by key in memory — simulating the Shuffle phase

③ MRJob calls reducer() for each unique key — simulating the Reduce phase

④ Results print to your terminal — you see the output immediately, no HDFS needed

# Step 6 — Run on Hadoop Cluster

You are now submitting to real distributed machines

```
# 1. Find the Hadoop streaming JAR
$ ls /usr/lib/hadoop-mapreduce/hadoop-streaming*.jar

# 2. Create unique output directory name (timestamp)
$ OUTDIR="lab5_out_$(date +%s)"

# 3. Run the job on the cluster
$ python3 src/mr_sales_per_category.py data/shopping_data_200.csv \
       -r hadoop  --hadoop-streaming-jar /usr/lib/hadoop-mapreduce/hadoop-streaming*.jar \
       --output-dir $OUTDIR  --python-bin python3
```

**Key flags explained:**

| | |
|---|---|
| `-r hadoop` | → Tells MRJob to use real Hadoop cluster instead of local mode — the KEY switch |
| `--hadoop-streaming-jar` | → Points to the JAR that bridges Python code and Java-based Hadoop streaming engine |
| `--output-dir $OUTDIR` | → HDFS location for results. Uses the variable set above (unique per run) |
| `--python-bin python3` | → Tells ALL cluster workers to use python3 (not python2) when running your script |
| `date +%s` | → Unix timestamp = seconds since 1970. Makes folder name unique every run (avoids HDFS conflict) |

# Step 7 — View & Retrieve Results from HDFS

HDFS is NOT your local filesystem — you need special "hadoop fs" commands to interact with it

```
$ hadoop fs -ls $OUTDIR
```

hadoop fs = Hadoop filesystem client. -ls lists files stored on HDFS in your output directory.

*Output is split across multiple part-00000, part-00001… files (one per reducer task). This shows them all.*

```
$ hadoop fs -getmerge $OUTDIR result.out
```

-getmerge downloads ALL part-xxxxx shards and merges them into ONE local file named result.out.

*Hadoop always produces multiple output files. getmerge combines them into one for easy viewing/submission.*

```
$ head result.out
```

Display first 10 lines of result.out — your final MapReduce output in (key, count) format.

*Quick sanity check — verify categories and counts look correct before submitting to Canvas.*

```
$ hadoop fs -rm -r $OUTDIR
```

-rm -r = remove recursively. Permanently deletes the output directory from HDFS.

*CRITICAL: Hadoop CANNOT overwrite an existing output dir. You MUST delete before re-running, or use a new folder name.*

# Expected Output

What you should see after running head result.out

```
"Grocery"	40
"Electronics"	38
"Stationery"	41
"Personal Care"	39
"Clothing"	42
```

`"Grocery"` **Key = category name**

The value yielded by your mapper. mrjob wraps string keys in quotes — this is normal and expected output.

`\t (tab)` **Tab separator**

Hadoop uses tab (\t) between key and value, not a space or comma. If you need to parse result.out, use .split("\t").

`40` **Value = count**

The sum of all 1s emitted for this category. 40 means there were 40 grocery purchases in the 200-row dataset.

`Order?` **Output is NOT sorted**

Hadoop does not guarantee order across categories. If you need sorted output, add a sort command after getmerge.

# Full Workflow — End to End

**GitHub Repo**
Store code + data

**git clone**
Download to cluster

**Linux CLI**
Inspect & explore

**Write code**
Create MR job

**Run locally**
Debug & verify

**Run Hadoop**
Distributed run

**HDFS output**
Results on cluster

**getmerge**
Retrieve & view

```
# Complete command sequence
git clone https://github.com/<you>/lab5-hpc-mapreduce.git
cd lab5-hpc-mapreduce
python3 src/mr_sales_per_category.py data/shopping_data_200.csv  # local test
OUTDIR="lab5_out_$(date +%s)"  &&  python3 src/mr_sales_per_category.py data/shopping_data_200.csv -r hadoop --hadoop-streaming-jar
/usr/lib/hadoop-mapreduce/hadoop-streaming*.jar --output-dir $OUTDIR --python-bin python3
hadoop fs -getmerge $OUTDIR result.out  &&  head result.out
```

# You Are Ready for HW3!

*This lab = mini version of Homework 3*

| | | | | |
|---|---|---|---|---|
| ✓ GitHub repos | ✓ Linux CLI | ✓ MapReduce code | ✓ Hadoop cluster | ✓ HDFS retrieval |