Map-Reduce with Python

MOD 2.1 — Big Data Challenges

Stéphane Derrode & Lamia Derrode
Centrale Lyon — Mathematics & Computer Science Dept.

Outline

1

Big Data & Parallelism

Why do we need a new programming model?

2

Python map() & reduce()

Functional programming foundations

3

Key-Value Pairs & the Pipeline

The Hadoop MapReduce model

4

Word Count — A Complete Example

From theory to Python code

Big Data: The Scale Challenge

Examples of daily data volumes (2024):

Meta (Facebook + Instagram + WhatsApp)	**~4 petabytes / day**
Google Search	**~8.5 billion queries / day**
Large Hadron Collider @ CERN	**~1 petabyte / day**
Global internet traffic	**~500 exabytes / day**

Capacity of a single (large) server:

CPU cores: 192 – 256

RAM: 2 Terabytes

Storage: 8 – 32 Terabytes

Disk speed: 100 – 400 GB/s

Solution: Massive Parallelism — distribute computation across thousands of commodity servers

The Parallelism Problem: Fault Tolerance

With thousands of servers, hardware failure is not the exception — it is the norm:

1 server	fails once every few months
1 000 servers	mean time to failure < 1 day
A large job	can take several days → a failure is guaranteed to occur

What we need:

- **Encapsulate parallelism** — the developer should not manage threads or processes
- **Automatic fault tolerance** — no manual checkpointing; failed tasks are re-run automatically
- **Write once, use by all** — coded once by experts, usable by non-experts

→ MapReduce: a simple programming model with built-in fault tolerance

The Hadoop Ecosystem

Inspired by Google's MapReduce paper (Dean & Ghemawat, 2004)

YARN	Resource Manager — schedules jobs across the cluster
MapReduce	Distributed computation ← our focus in this course
HDFS	Distributed File System — stores data across nodes

Apache Hadoop is an open-source project: hadoop.apache.org

Modern successor — Apache Spark (2014)

In-memory processing, up to 100× faster than Hadoop MapReduce. Keeps the key-value paradigm but adds a richer API (DataFrames, MLlib, Spark Streaming). Widely used in industry today.

Part 2 — Python map() & reduce()

Functional programming foundations

The map() Function

map(function, iterable) — applies a function independently to every element

```
# Syntax
result = list(map(function, iterable))

# Example 1: square each number
numbers = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x**2, numbers))
# → [1, 4, 9, 16, 25]

# Example 2: convert strings to uppercase
words = ['hello', 'world', 'python']
upper = list(map(str.upper, words))
# → ['HELLO', 'WORLD', 'PYTHON']
```

Question: How to convert ['1', '2', '3', '4'] into [1, 2, 3, 4] using map()?

→ `list(map(int, ['1', '2', '3', '4']))`

The reduce() Function

reduce(function, iterable) — aggregates all elements into a single result

```
from functools import reduce   # required in Python 3

# Example 1: sum all elements
numbers = [1, 2, 3, 4, 5]
total = reduce(lambda acc, x: acc + x, numbers)
# → 15    computed as  ((((1+2)+3)+4)+5)

# Example 2: find the maximum
values = [3, 1, 4, 1, 5, 9, 2, 6]
maxi = reduce(lambda a, b: a if a > b else b, values)
# → 9
```

How reduce() processes [1, 2, 3, 4, 5] with (+):

```
1 + 2 → 3    ▶    3 + 3 → 6    ▶    6 + 4 → 10    ▶    10 + 5 → 15
```

→ The function is applied cumulatively, left to right, until one value remains.

Combining map() and reduce()

Together they form a complete data processing pipeline:

```
from functools import reduce

# Goal: compute the total number of characters in a list of words
words = ['Big', 'Data', 'Map', 'Reduce']

# Step 1 — MAP: compute the length of each word
lengths = list(map(len, words))
# → [3, 4, 3, 6]

# Step 2 — REDUCE: sum all lengths
total = reduce(lambda a, b: a + b, lengths)
# → 16

# In one line:
total = reduce(lambda a, b: a + b, map(len, words))
```

['Big','Data',
'Map','Reduce']

—map(len)—▶

[3, 4, 3, 6]

—reduce(+)—▶

16

Part 3 — Key-Value Pairs & the Pipeline

The Hadoop MapReduce model

Key-Value Pairs in Hadoop

In Hadoop, all data is represented as (key, value) pairs — keys and values can be of any type.

Key type	Value type	Example pair
Text (str)	Integer (int)	`('hello', 17)`
Integer	Tuple	`(17, ('hello', 3))`
Integer	Text (str)	`(1, "Two roads diverged in a yellow wood")`
Text (str)	Float	`('Paris', 2.3)`

"The Road Not Taken" — Robert Frost (paragraph index as key):

```
(1, "Two roads diverged in a yellow wood")
(2, "And sorry I could not travel both")
(3, "And be one traveler, long I stood")
```

Map & Reduce on (Key, Value) Pairs

Map: f(key, value) → list of (key, f(value)) [applied independently to each pair]

```
[('Home',1), ('Garden',3), ('Park',1)]  →  [('Home',f(1)), ('Garden',f(3)), ('Park',f(1))]
```

Reduce: f(key, list(values)) → (key, result) [applied to all values sharing the same key]

```
[('Home',1), ('Home',3), ('Home',1)]  ==  ('Home',[1,3,1])  →  ('Home', f([1,3,1]))
```

The Hadoop MapReduce pipeline:

(key, value) pairs flow through each stage. The Shuffle & Sort phase is handled automatically.

Key insight: you only write the Mapper and the Reducer.

Hadoop handles distribution, parallelism, fault tolerance, and shuffling automatically.

Part 4 — Word Count: A Complete Example

From theory to Python code

Word Count — Step 1: Mapping

For each word in the text, the mapper emits a (word, 1) pair:

Input text:

```
"Dear Brutus"
"Et tu Brutus"
"Brutus is brave"
```

mapper
(word, 1)

Mapper output:

```
('Dear', 1)
('Brutus', 1)
('Et', 1)
('tu', 1)
('Brutus', 1)
('Brutus', 1)
('is', 1)
('brave', 1)
```

Notice: "Brutus" appears 3 times → 3 separate ('Brutus', 1) pairs. No aggregation yet.

Word Count — Step 2: Shuffling & Sorting

The framework groups all pairs by key and sorts them — automatically:

Before (mapper output):

```
('Dear', 1)
('Brutus', 1)
('Et', 1)
('tu', 1)
('Brutus', 1)
('Brutus', 1)
('is', 1)
('brave', 1)
```

sort
by key

After (sorted by key):

```
('Brutus', 1)
('Brutus', 1)
('Brutus', 1)
('Dear', 1)
('Et', 1)
('brave', 1)
('is', 1)
('tu', 1)
```

→ All "Brutus" pairs are now contiguous — the reducer can aggregate them in a single pass.

Word Count — Step 3: Reducing

The reducer sums all values for the same key — one pass through the sorted list:

Sorted input:

```
('Brutus', 1)
('Brutus', 1)
('Brutus', 1)
('Dear', 1)
('Et', 1)
('brave', 1)
('is', 1)
('tu', 1)
```

reduce
(sum)

Final output:

```
('Brutus', 3)
('Dear', 1)
('Et', 1)
('brave', 1)
('is', 1)
('tu', 1)
```

"Brutus" counted 3 times: 1 + 1 + 1 = 3. This only works because the input was sorted.

Word Count in Python — Mapper

Reads text from standard input, emits (word, 1) pairs to standard output:

```
import sys

# Input comes from STDIN (standard input)
for line in sys.stdin:
    line = line.strip()       # remove leading/trailing whitespace
    words = line.split()      # split line into words
    for word in words:
        # Write (word, 1) pairs to STDOUT — tab-delimited
        print(f"{word}\t1")
```

Running the full pipeline locally (Unix pipes):

```
$ cat dracula | python mapper.py                    # step 1: map
$ cat dracula | python mapper.py | sort             # step 2: sort
$ cat dracula | python mapper.py | sort | python reducer.py  # full pipeline
```

Word Count in Python — Reducer

Reads sorted (word, 1) pairs from stdin, sums counts per word:

```
import sys
current_word, current_count = None, 0

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:          # same key → accumulate
        current_count += count
    else:                             # new key → emit previous result
        if current_word is not None:
            print(f"{current_word}\t{current_count}")
        current_count = count
        current_word = word

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

⚠ **Classic bug: forgetting to emit the last word after the loop.**

Running the Pipeline with wordcount_mapreduce.py

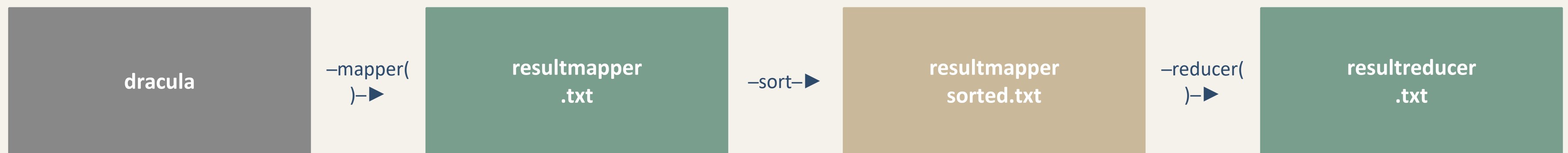
In the lab, you work with a self-contained Python script that wraps the 3 steps:

```
# Step 1 — MAPPER
mapper('dracula', 'resultmapper.txt')

# Step 2 — SORT  (emulates Hadoop's Shuffle & Sort)
os.system('sort resultmapper.txt -o resultmappersorted.txt')

# Step 3 — REDUCER
reducer('resultmappersorted.txt', 'resultreducer.txt')
```

Pipeline:



Tip for the lab: to adapt this script to a new problem, only modify the mapper() and reducer() functions and update the filenames. The pipeline structure stays identical.

Quick Check

In the MapReduce pipeline, what is the role of the Shuffle & Sort phase?

A

It applies the mapper function in parallel across all nodes

B

It groups all values with the same key before they reach the reducer

C

It writes the final results to disk

D

It splits the input file into chunks for parallel processing

Summary & What's Next

MapReduce in a nutshell

A 3-phase pipeline: Map → Shuffle/Sort → Reduce. You write 2 functions; the framework handles the rest.

The key-value paradigm

All data flows as (key, value) pairs. Shuffle & Sort groups values by key before reducing.

Python implementation

wordcount_mapreduce.py emulates the pipeline locally: mapper() → os.system(sort) → reducer().

Beyond Hadoop

Apache Spark extends this model with in-memory processing and a richer API (DataFrames, MLlib).

Now let's practice! → TP 2: Map-Reduce with Python