BSC « BIG-DATA COMPUTING TECHNOLOGIES », CHAPTER 2

Stéphane Derrode, Centrale Lyon
sephane.derrode@ec-lyon.fr

Chapter 2 outline

- ***Lecturers***
  Lamia Derrode & Stéphane Derrode (Centrale Lyon).

- ***Time allocation*** 28h,
  including 1h for mid-term exam and 2h for project restitution.

- ***Organisation***
  - Part 1. Linked Open Data (LOD) technology (6h) and project (7h).
  - Part 2. Hadoop framework, including HDFS and MrJob library (8h).
  - Part 3. Spark framework, using pyspsark python library (4h).

- ***Assessment***
  - Lab report (Part 2): 20% of the final grade.
  - LOD project (Part 1): 40% of the final grade: 20% for the report and 20% for the project defense.
  - Exam: Mid-term exam (Chapter 1 (MongoDb), and Chapter 2 (Part 1. and Part 2.)) will account for 20% of the final grade. (February 18th, 2025)

- ***Detailled content***
  http://perso.ec-lyon.fr/derrode.stephane/Teaching/BSC/chapter2/

Hadoop motivations (1/2)

Examples:

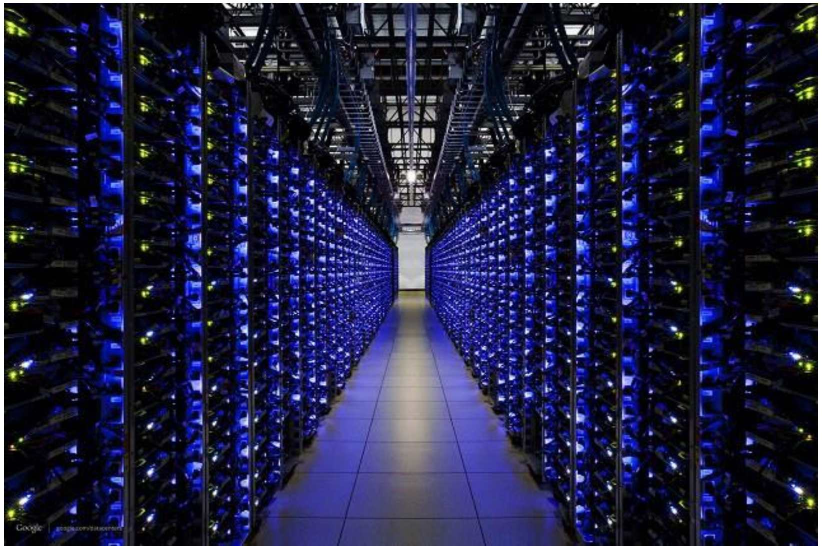
- Google, 2008: 20 PB / day, 180 GB / job
- Web index: 50 billion pages, 15 PB
- Large Hadron Collider (LHC)@CERN: 15 PB / year

Capacity of a (large) server:

- RAM: 256 GB
- HDD: 24 TB
- HDD transfer speed: 100 MB/s

Solution: Parallelism

- Hadoop cluster @ Yahoo: 4000 servers, reading the web in parallel takes about 1h20



Google data center

Hadoop motivations (2/2)

- **The parallelism problem**
  - 1 server might crash every few months.
  - 1000 servers → average time before a crash is less than 1 day
- **A "big" job can take several days**
  - Hardware failure: this is normal!
  - Parallelism: impossible to resume partially in case of failure (checkpointing and replication are difficult to implement correctly).
- **Big Data Platforms**: everyone should be able to write programs
  - Encapsulates parallelism
  - Encapsulates fault tolerance
  - Written once by experts, beneficial for all (non-experts, that is to say "us")

CHAPTER 2

PART 2.1 (4H) – HADOOP FRAMEWORK

1. Map & reduce functions in Python
2. Hadoop
   1. Hadoop map-reduce
   2. Hadoop & HDFS
3. Introduction to lab

1. Map & reduce functions in Python

Two very simple functions inspired by functional programming.

MAP function

Transformation: `map`

`map(f, [x1, …, xn]) = [f(x1), …, f(xn)]`

Example:

python Copier

```
map(lambda x: 2 * x, [1, 2, 3]) → [2, 4, 6]
```

In Python:

python Copier

```
# mapexemple.py
print(list(map(lambda x: 2 * x, [1, 2, 3])))  # Output: [2, 4, 6]
```

1. Map & reduce functions in Python

Two very simple functions inspired by functional programming.

REDUCE function

Aggregation: `reduce`

`reduce(f, [x1, …, xn]) = f(x1, f(x2, …, f(xn-1, xn)))`

Example:

```python
reduce(lambda x, y: x + y, [2, 4, 6]) → 12
```

In Python:

```python
# reduceexemple.py
from functools import reduce
print(reduce(lambda x, y: x + y, [2, 4, 6]))  # Output: 12
```

1. Map & reduce functions in Python

Two very simple functions inspired by functional programming.

These functions are generic because they take a function as a parameter: the developer provides the functions.

Example 1: `map` with a function:

python Copier

```python
map(str.upper, ['hello', 'data']) → ['HELLO', 'DATA']
```

Example 2: `reduce` with a function:

python Copier

```python
reduce(max, [3, 45, 27]) → 45
```

1. Map & reduce functions in Python

MAP examples

Example 3: Convert temperatures from Celsius to Fahrenheit

python Copier

```python
celsius = [0, 10, 20, 30, 40]
fahrenheit = list(map(lambda x: (x * 9/5) + 32, celsius))
print(fahrenheit)  # Output: [32.0, 50.0, 68.0, 86.0, 104.0]
```

Example 4: Add two lists element-wise

python Copier

```python
list1 = [1, 2, 3]
list2 = [4, 5, 6]
summed_lists = list(map(lambda x, y: x + y, list1, list2))
print(summed_lists)  # Output: [5, 7, 9]
```

Example 5: Extract lengths of strings

python Copier

```python
words = ['apple', 'banana', 'cherry']
lengths = list(map(len, words))
print(lengths)  # Output: [5, 6, 6]
```

Question: How to convert a list of string such as ['1', '2', '3', '4'] into a list of int ?

1. Map & reduce functions in Python

REDUCE examples

Example 6: Flatten a list of lists

python

Copier

```
from functools import reduce

list_of_lists = [[1, 2], [3, 4], [5, 6]]
flattened_list = reduce(lambda x, y: x + y, list_of_lists)
print(flattened_list)  # Output: [1, 2, 3, 4, 5, 6]
```

Example 7: Calculate the greatest common divisor (GCD) of a list of numbers

python

Copier

```
from functools import reduce
import math

numbers = [24, 36, 48]
gcd_of_numbers = reduce(math.gcd, numbers)
print(gcd_of_numbers)  # Output: 12
```

1. Map & reduce functions in Python

BOTH functions

python Copier

```
from functools import reduce

# List of numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Step 1: Use map to square only even numbers
squared_evens = map(lambda x: x ** 2 if x % 2 == 0 else 0, numbers)

# Step 2: Use reduce to sum the squared even numbers
sum_of_squares = reduce(lambda x, y: x + y, squared_evens)

# Print the result
print("Sum of squares of even numbers:", sum_of_squares)
```

2. Hadoop

Hadoop in 5 minutes / simplilearn

2.1 Hadoop map-reduce

Data in Hadoop is always considered as key-value pairs.

A key can be of any type.

- In pair (‘Hello’, 17)
  - ‘Hello’ is the key (text)
  - 17 is the value (int)
- In pair (17, ('Hello', 3))
  - 17 is the key (int)
  - ('Hello', 3) is the value (tuple)

When working with a book, each paragraph to be processed is numbered. In this case, the key is the paragraph number, and the value is the paragraph itself:

(1, "Two roads diverged in a yellow wood")
(2, "And sorry I could not travel both")
(3, "And be one traveler, long I stood")
(4, "And looked down one as far as I could")
(5, "To where it bent in the undergrowth;")

"The Road Not Taken" from Robert Frost

2.1 Hadoop map-reduce

How map & reduce functions apply for (key-value) pairs?

- **Map:** Function *f* is applied to each pair **independently**.

  f(key, value) → list(key, f(value))

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

- **Reduce:** Function *f* is applied to all values with **the same key**.

  f(key, list(value)) → (key, f(list(value)))

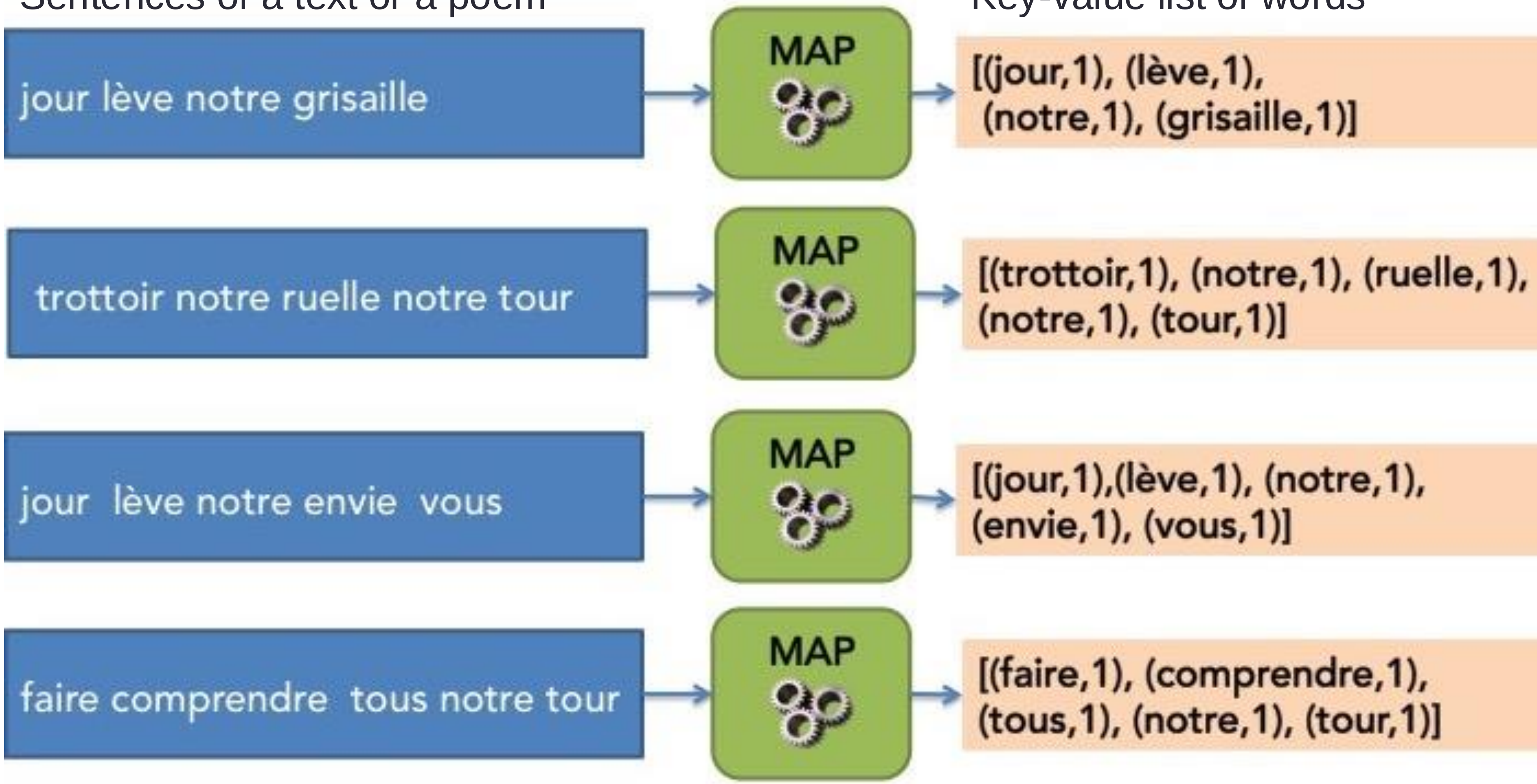
  [('Home', 1), ('Home', 3), ('Home', 1)] == ('Home', [1,3,1])
  → ('Home', f([1,3,1]))

The basic hadoop map-reduce process works like this

2.1 Ex: counting the frequency of a word

Sentences of a text or a poem

Key-value list of words



Parallel processing
(e.g. the cores of a processor)

2.1 Ex: counting the frequency of a word

Shuffling & sorting

(comprendre, [1])

(envie,[1])

(faire,[1])

(grisaille,[1])

(jour,[1,1])

(lève, [1,1])

(notre, [1,1,1,1,1])

(ruelle,[1])

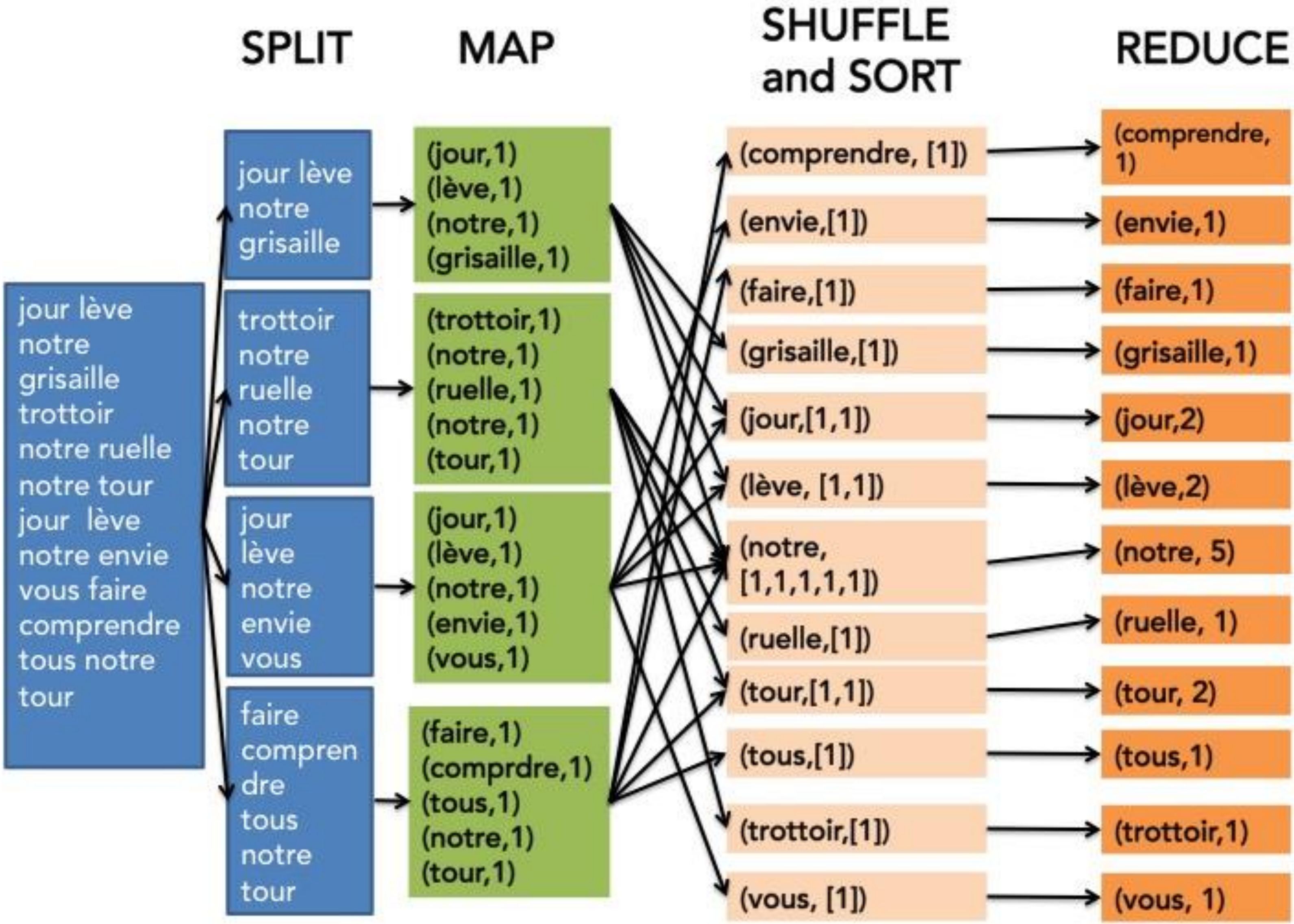
(tour,[1,1])

(tous,[1])

(trottoir,[1])

(vous, [1])

2.1 Ex: counting the frequency of a word



2.1 Word count in Python (map)

```
#!/usr/bin/env python3
#file wc_mapper.py

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
   # remove leading and trailing whitespace
   line=line.strip()
   # split the line into words
   words=line.split()
   # increase counters
   for word in words:
       # write the results to STDOUT (standard output);
       # what we output here will be the input for the
       # Reduce step, i.e. the input for reducer.py
       # tab-delimited; the trivial word count is 1
       print(word,'\t1')
```

2.1 Word count in Python (reduce)

```
#!/usr/bin/env python3
#file wc_reducer.py
import sys

current_word=None
current_count=0
word=None
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(current_word, '\t', current_count)
       current_count=count
       current_word=word

if current_word==word:
   print(current_word, '\t', current_count)
```

Step #1 of the Lab to be completed by students

2.2 Hadoop & HDFS

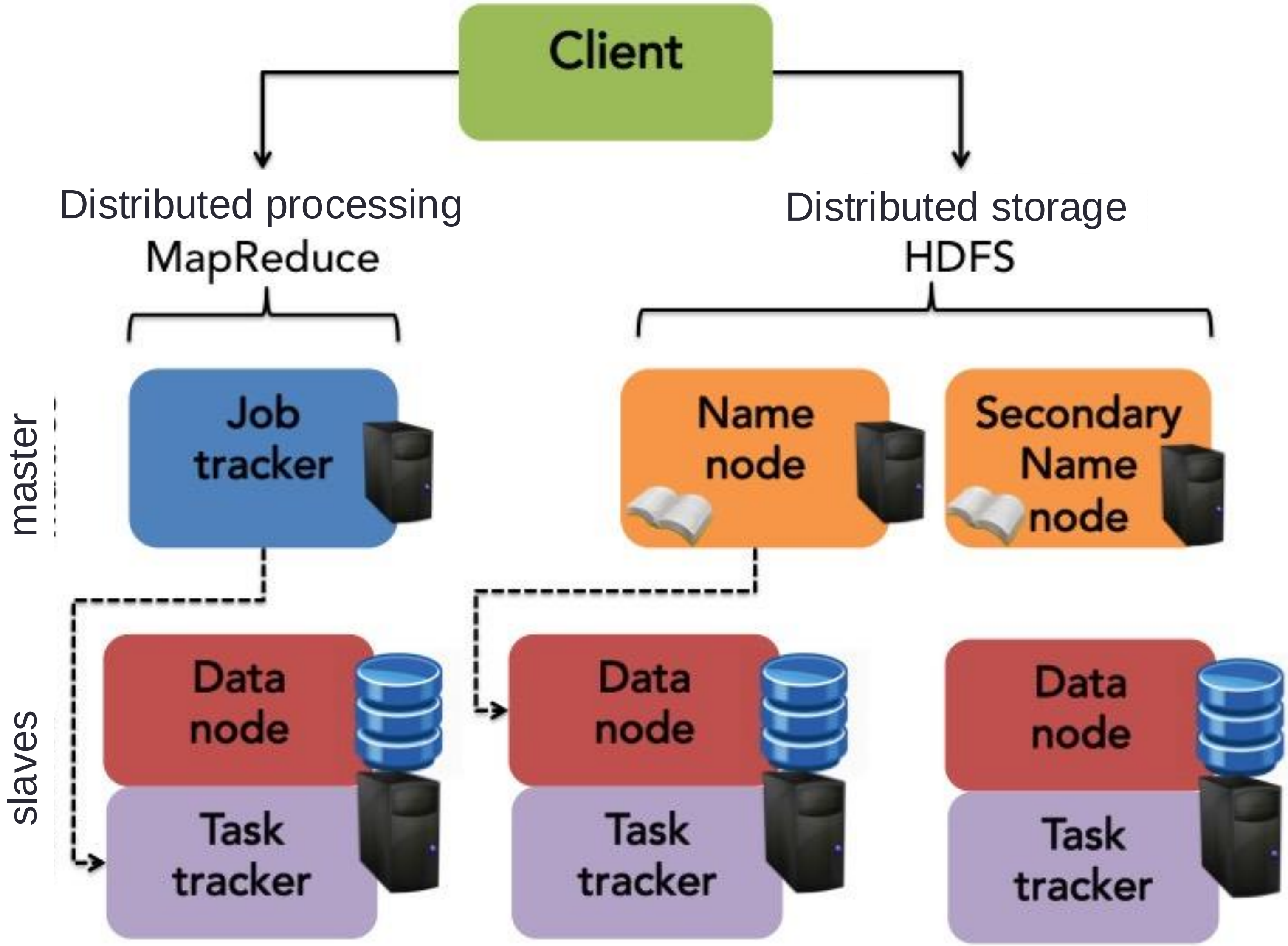
The technical foundation of Hadoop consists of:

- All the necessary support architecture for orchestrating MapReduce, which includes:
  - Job scheduling,
  - File location,
  - Execution distribution.

- A HDFS (Hadoop Distributed File System) that is:
  - Distributed: data is spread across the machines in the cluster.
  - Replicated: in case of failure, no data is lost.
  - Optimized for the co-location of data and processing.

Section heavily based on [openclassroom]

2.2 Hadoop & HDFS



2.2 HDFS

- **Objectives of the Distributed File System:**
  - Fault-tolerant (redundancy)
  - High-performance (parallel access)

- **Large Files**
  - Sequential read and write

- **Data Processing "at the closest"**
  - Data is stored on the machines that process it
  - For better resource utilization of machines
  - To avoid network transfers (latency)

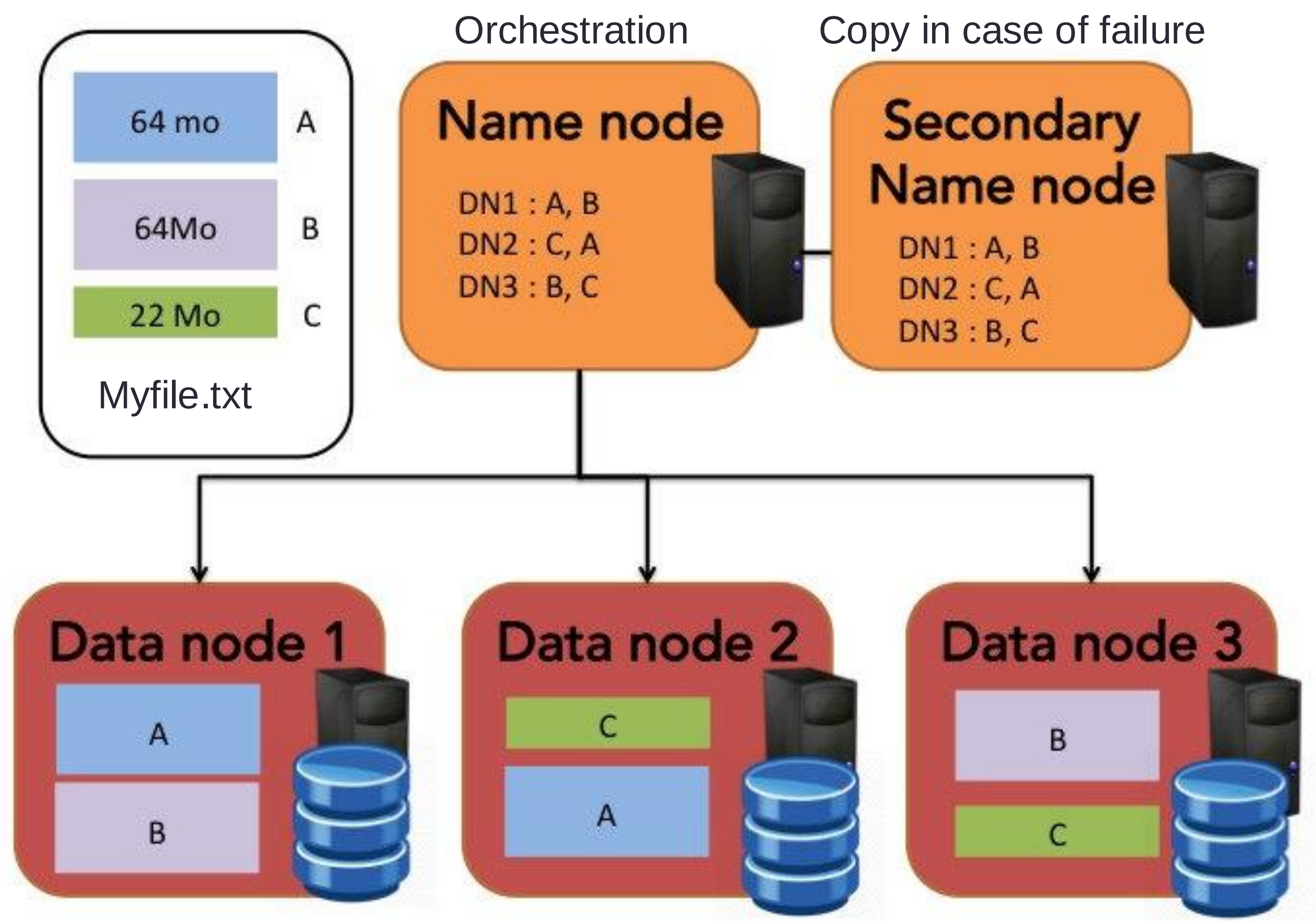
- **Data is organized in files and directories**
  - Mimics standard file management systems
  - Files are split into blocks (64MB) and distributed across servers with replication (3 times by default)
  - Whenever possible, process data on the machines where it is stored.

2.2 « master / slave » architecture

- A master: the « NameNode »
  - Manages file names, access rights, etc.
    - Stores metadata associated with files.
    - Keeps everything in RAM (maximum: 60M objects and 16GB).
  - Oversees operations on files and blocks.
  - Monitors the health of the system (failures, crashes), and load balances.

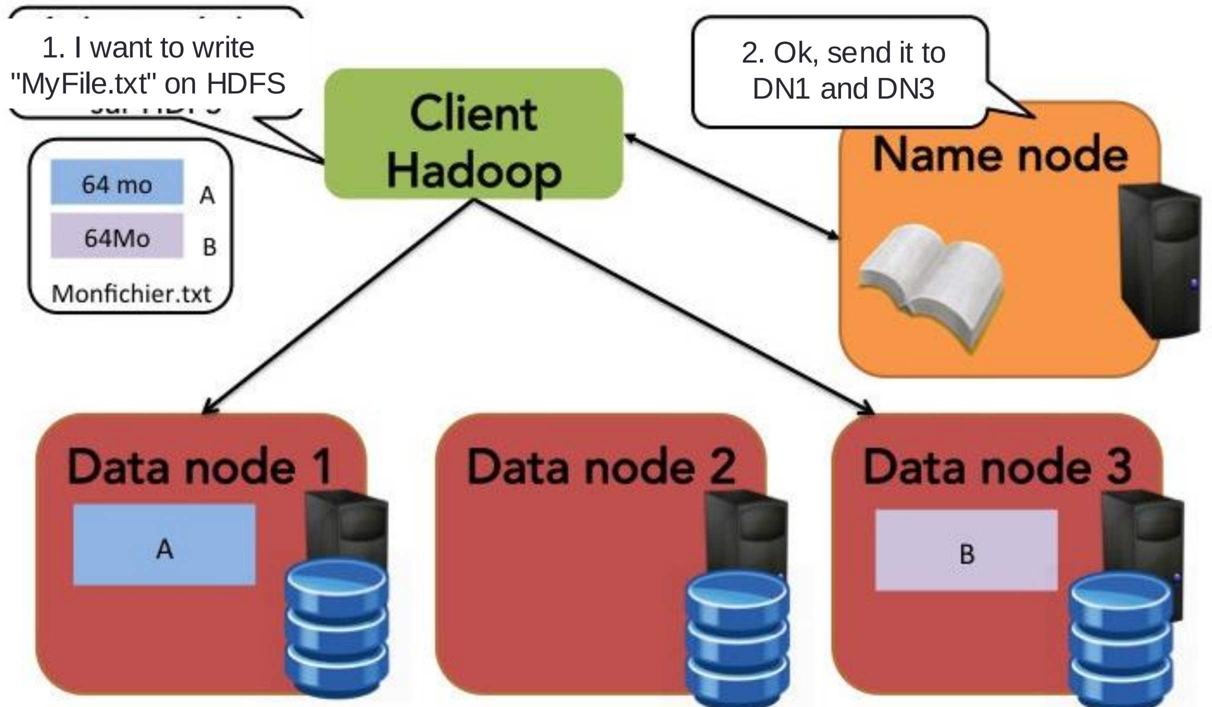
- Thousands of slaves: the « DataNodes »
  - Stores data (blocks).
    - Data never passes through the **NameNode**.
  - Performs read and write operations.
  - Performs copies (replications) ordered by the **NameNode.**
  - Regularly checks the health of the **NameNode**.
  - Reports to the **NameNode** if any blocks are corrupted (checksum).

2.2 « master / slave » architecture



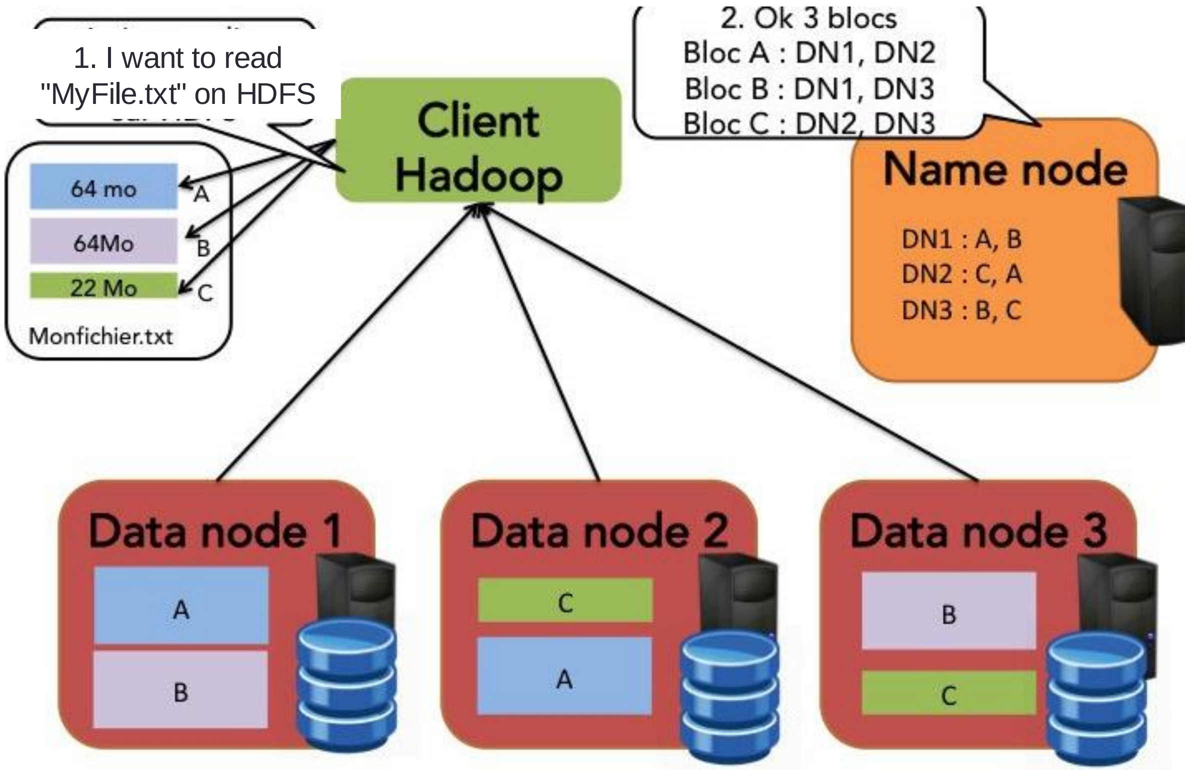
2.2 Copy a file to HDFS

1. The client tells the **NameNode** that it wants to write a block.
2. The **NameNode** indicates which **DataNode** to contact.
3. The client sends the block to the **DataNode**.
4. The **DataNodes** replicate the blocks among themselves.
5. The cycle repeats for the next block.

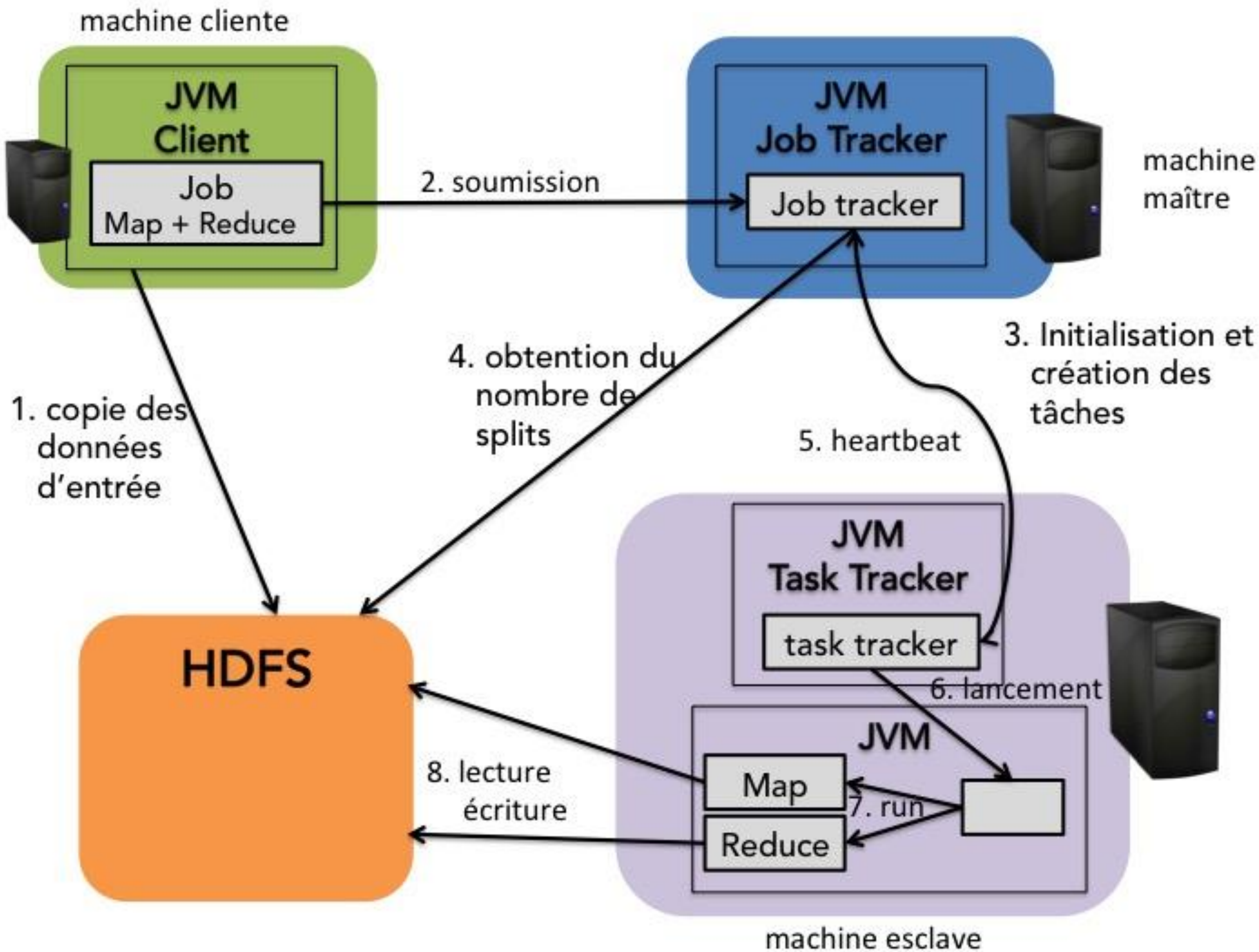


2.2 Read a file on HDFS

1. The client tells the **NameNode** that it wants to read a file.
2. The **NameNode** provides the file's size and the **DataNodes** containing the blocks.
3. The client retrieves each block from one of the **DataNodes**.
4. If a **DataNode** is unavailable, the client contacts another one.



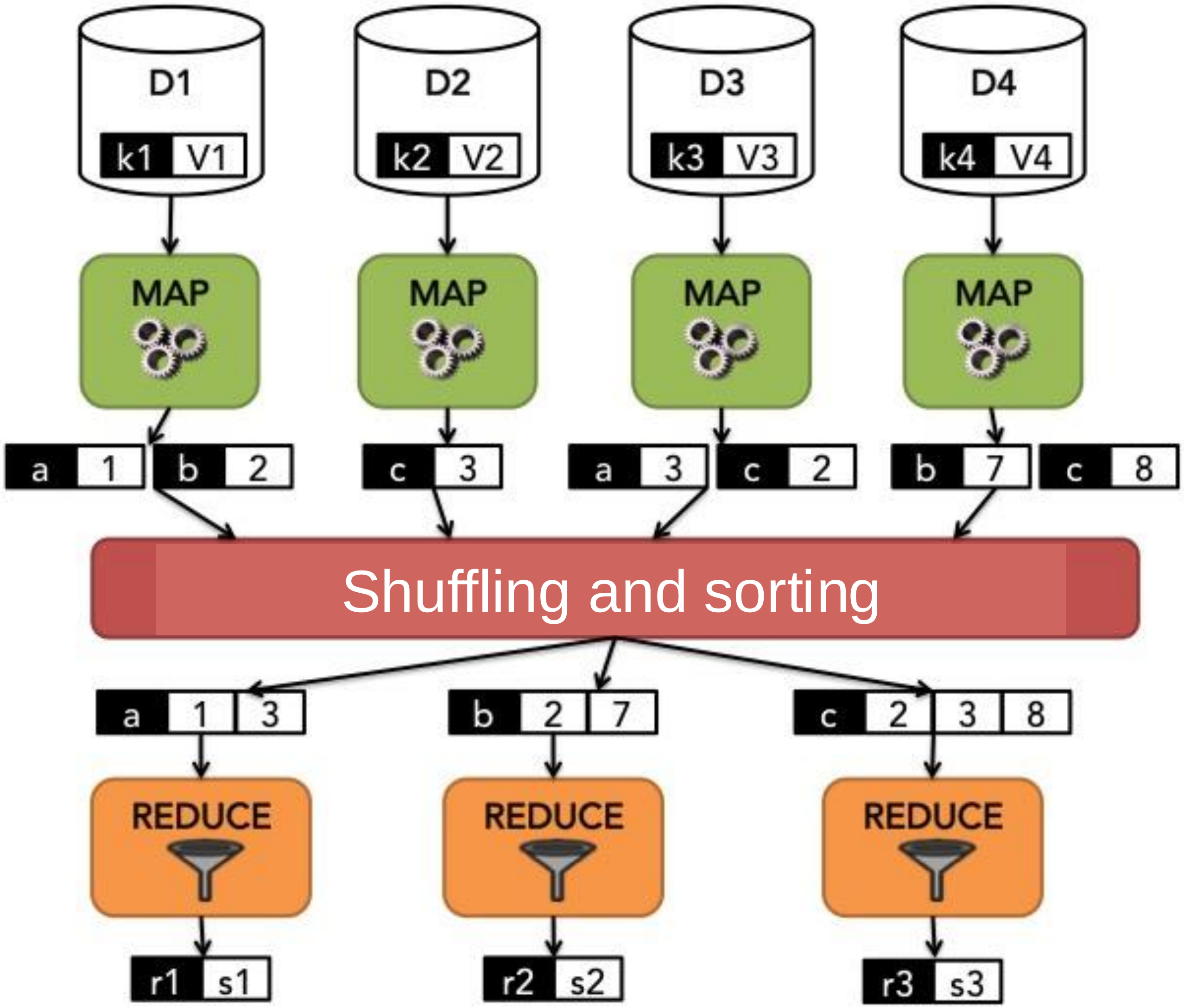
2.2 Submitting a job in Hadoop



JVM : Java virtual Machine.

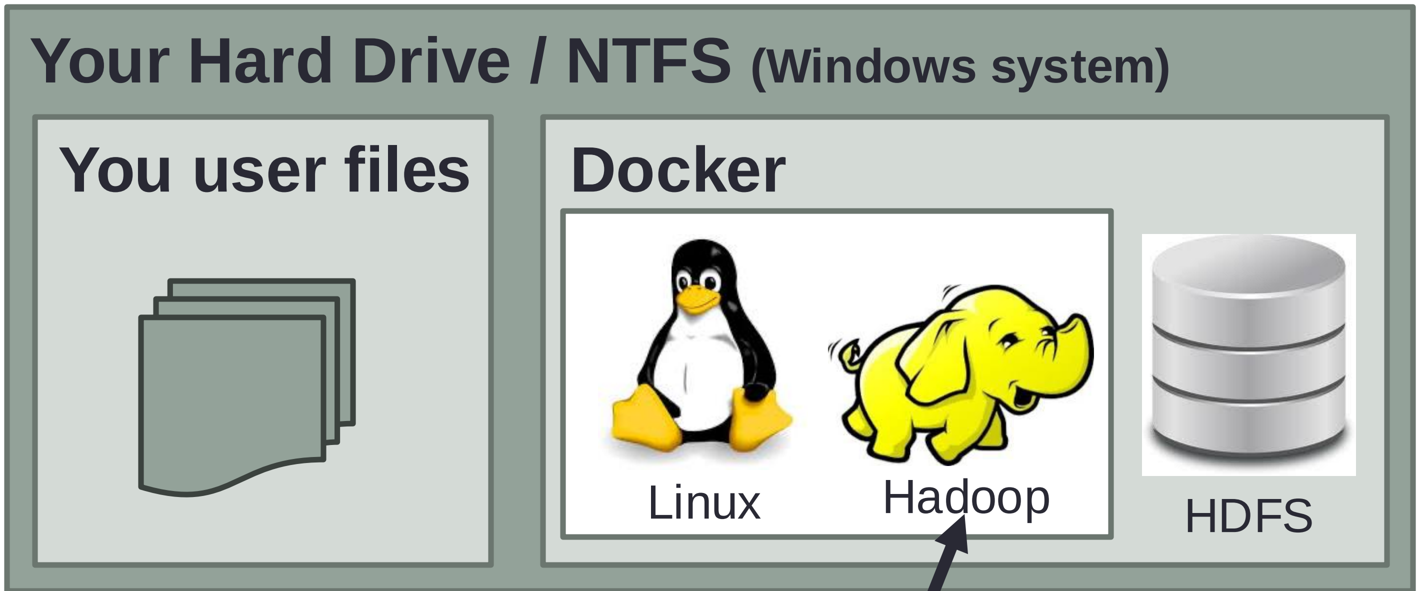
Remember that Hadoop is natively developed in Java!

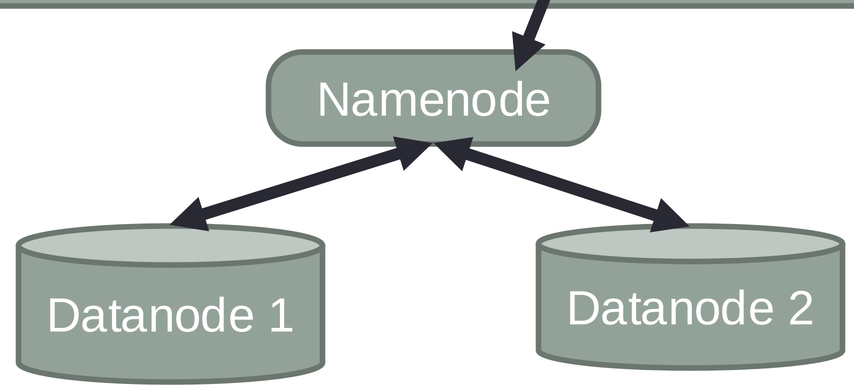
2.2 Map-reduce in Hadoop



3. Introduction to lab : Docker with hadoop container

Your personnal computer (Windows)





Linux : Only works on command line: there si no user graphical interface to manipulate files.

Terminals : One Terminal should focus on your own system, another one on the Linux system inside the container. Both terminals have to communicate.

CHAPTER 2

PART 2.2 (4H) – MRJOB LIBRARY

1. Iterators & Generators
2. MrJob library

1. Iterators & Generators in Python

An **iterator** is a type of cursor whose task is to move through a sequence of objects.
The iterator allows you to traverse each object in a sequence without worrying about the underlying structure.

A list and a list comprehension are iterables, not iterators

```
# a list is an iterable
liste=[1,2,3,4,5,6,7,8,9,10]
for x in liste:
   print(x)


# A comprenhension list is also an iterable
a_list=[1,9,8,4]
A=[elem*2 for elem in a_list]
print(A)
```

1.1 Iterators

In Python, an **iterator** is an object that implements two essential methods:

1. **__iter__():** This method returns the iterator object itself. It's used to initialize the iterator and is required to make the object iterable.
2. **__next__():** This method returns the next item in the sequence. When there are no more items to return, it raises the **StopIteration** exception to signal the end of the iteration.

```python
class Countdown:
    def __init__(self, start):
        self.start = start
        self.current = start

    def __iter__(self):
        # The iterator object itself is returned by __iter__()
        return self

    def __next__(self):
        # Return the next number in the sequence
        if self.current <= 0:
            raise StopIteration  # Stop the iteration when we reach 0
        self.current -= 1
        return self.current + 1  # Return the current value

# Creating an instance of the Countdown iterator
countdown = Countdown(5)

# Using the iterator in a loop
for num in countdown:
    print(num)
```

1.1 Iterators

```
class Fibonacci:
    def __init__(self, max_value):
        self.max_value = max_value
        self.a, self.b = 0, 1  # Initializing the first two Fibonacci numbers

    def __iter__(self):
        return self  # The iterator object itself

    def __next__(self):
        if self.a > self.max_value:
            raise StopIteration  # Stop when the value exceeds the max_value
        current_value = self.a
        self.a, self.b = self.b, self.a + self.b  # Update to the next Fibonacci
        return current_value

# Creating an instance of the Fibonacci iterator that generates Fibonacci number
fib = Fibonacci(100)

# Using the iterator in a loop
for number in fib:
    print(number)
```

1.2 Generators

Generator: A simpler and memory-efficient way to create an **iterator** using the **yield** keyword, which generates values lazily.
The keyword **yield** is somewhat similar to the **return** statement in functions, except that it doesn't signify the end of the function's execution. Instead, it pauses the function, and on the next iteration, the function will resume and look for the next **yield**.

```python
def simple_generator():
    yield 1
    yield 2
    yield 3

# Using the generator
for value in simple_generator():
    print(value)
```

```python
def count_up_to(max):
    count = 1
    while count <= max:
        yield count  # Yield the current value and pause
        count += 1

# Creating an iterator using the generator function
gen = count_up_to(3)

# Iterating over the generator
for num in gen:
    print(num)
```

1.2 Generators

Generator: A simpler and memory-efficient way to create an **iterator** using the **yield** keyword, which generates values lazily.
The keyword **yield** is somewhat similar to the **return** statement in functions, except that it doesn't signify the end of the function's execution. Instead, it pauses the function, and on the next iteration, the function will resume and look for the next **yield**.

```python
def fibonacci(max_value):
    a, b = 0, 1  # Initial Fibonacci numbers
    while a <= max_value:
        yield a  # Yield the current Fibonacci number
        a, b = b, a + b  # Update to the next Fibonacci numbers

# Using the Fibonacci generator
for number in fibonacci(100):
    print(number)
```

1.3 Genertaors in map-reduce scripts

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

# Generator function to yield words from each line
def word_generator(line):
    for word in line.strip().split():
        yield word.lower()  # Yield each word in lowercase to handle case insens

# Read from standard input (Hadoop streaming passes input to the script via stdi
for line in sys.stdin:
    # Yield each word from the line using the generator
    for word in word_generator(line):
        # Output word with count 1
        print(f"{word}\t1")
```

2. MrJob library

The **mrjob** library is a Python package that simplifies writing and running **MapReduce** jobs on **Hadoop** or **Amazon EMR (Elastic MapReduce)**. It provides an easy-to-use interface for creating and executing MapReduce jobs without having to deal with the low-level details of Hadoop's infrastructure.

Key Features of mrjob:

1. **Simplicity**: mrjob allows you to write MapReduce jobs in pure Python, eliminating the need to write complex Java code for Hadoop.
2. **Multiple Backends**: You can run jobs locally (on your machine), on a Hadoop cluster, or on **Amazon EMR**, making it highly flexible.
3. **Streaming Support**: It supports both **Hadoop Streaming** (for running MapReduce jobs on a Hadoop cluster) and local execution, where you can test your code without needing a cluster.
4. **Job Configuration**: It handles job configuration, input, output, and the connection to a cluster, making it simpler to focus on the logic of your MapReduce tasks.
5. **Pythonic Interface**: Instead of requiring the user to work with Java's MapReduce API, mrjob lets you write Mappers and Reducers as simple Python classes and functions.

2. MrJob library

```python
from mrjob.job import MRJob

# Define the MapReduce job
class MRWordCount(MRJob):

    # Mapper: Extracts each word from the line and outputs (word, 1)
    def mapper(self, _, line):
        for word in line.split():
            yield word.lower(), 1

    # Reducer: Sums the counts for each word
    def reducer(self, word, counts):
        yield word, sum(counts)

if __name__ == '__main__':
    MRWordCount.run()
```

To run the job locally on your machine (without a Hadoop cluster):

bash Copier

```bash
python mr_word_count.py input.txt
```

- `input.txt` is your input file containing the text.
- The output will be printed to the termin r saved as defined in the job.

For example, to run on **Amazon EMR**:

bash Copier

```bash
python mr_word_count.py -r emr s3://your-bucket/input.txt
```

2. MrJob library

With extra mapper_final method

```python
from mrjob.job import MRJob

class MRWordCountWithFinal(MRJob):

    def mapper(self, _, line):
        # Process each word from the line and count occurrences
        for word in line.split():
            yield word.lower(), 1  # Emit word with count 1

    def mapper_final(self):
        # This method is called after all input lines are processed
        # We can use it to emit final computations for the mapper
        yield "mapper_done", "Finished processing all input lines"

    def reducer(self, key, values):
        if key == "mapper_done":
            # The reducer handles the final message from the mapper
            yield key, "Mapper completed its task"
        else:
            # Normal word count reduction
            yield key, sum(values)

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

2. MrJob library

```python
from mrjob.job import MRJob

class MRWordCountWithThreshold(MRJob):

    def __init__(self, *args, **kwargs):
        # Initialize the threshold for word count filtering
        super(MRWordCountWithThreshold, self).__init__(*args, **kwargs)

    def mapper(self, _, line):
        # Process each word from the line and yield each word with a count of 1
        for word in line.split():
            yield word.lower(), 1  # Yield word with count 1

    def reducer(self, key, values):
        # Sum the counts for each word and apply the threshold filter
        total_count = sum(values)
        if total_count >= self.options.threshold:
            yield key, total_count  # Only yield words that meet the threshold

    def configure_args(self):
        # Configure command-line options if needed
        super(MRWordCountWithThreshold, self).configure_args()
        # Example: You could add additional arguments to adjust the threshold dynamically
        self.add_passthru_arg('--threshold', type=int, default=5, help="Minimum count threshold for words")

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

```
python mr_word_count_with_threshold.py --threshold 2 input.txt
```

2. MrJob library

```
from mrjob.job import MRJob
from mrjob.step import MRStep

class MRWordCountAndFilter(MRJob):

    def mapper(self, _, line):
        # Step 1: Process each word from the line and yield each word with a count of 1
        for word in line.split():
            yield word.lower(), 1  # Yield word with count 1

    def reducer(self, key, values):
        # Step 1: Sum the counts for each word
        total_count = sum(values)
        yield key, total_count  # Emit the word and its total count

    def filter_mapper(self, word, count):
        # Step 2 Mapper: Process the output of the previous step and filter words by length
        if len(word) >= 4:  # Only consider words that have 4 or more characters
            yield word, count

    def steps(self):
        # Define the steps using MRStep:
        # Step 1: Count words
        # Step 2: Filter out words shorter than 4 characters
        return [
            MRStep(mapper=self.mapper, reducer=self.reducer),  # Step 1: Word count
            MRStep(mapper=self.filter_mapper)  # Step 2: Filter words by length
        ]

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