BUSINESS INTELLIGENCE AND DATA ANALYTICS
– WE PREDICT THE FUTURE –

University of Petra

Institute of Analytics

University of Petra

307401
Big Data and Data Warehouses

Practical Examples

Simple Map and reduce in Python (No Big Data)

The Map Function

This simple python code (no big data) demonstrate the idea of map where we map a function to a list of numbers.

- **Method 1:**

```
lst = [1, 2, 3, 4]
list(map(lambda x: x*x, lst))
```

```
[1, 4, 9, 16]
```

- **Method 2:**

```
def square(x):
        return x*x
list(map(square, lst))
```

```
[1, 4, 9, 16]
```

Simple Map and reduce in Python (No Big Data)

The Reduce Function

This simple python code (no big data) demonstrate the idea of reduce, where we reduce a group of numbers into a single number.

- **Method 1:**

```
from functools import reduce
reduce(lambda x, y: x + y, lst)
10
```

- **Method 2:**

```
def add_reduce(x, y):
    out = x + y
    print(f"{x}+{y}-->{out}")
    return out
reduce(add_reduce,lst)
```

3<--2+1
6<--3+3
10<--4+6
10

Map Reduce Using MRJob Python Library

MapReduce refresher

word count is:
Map: split → (word, 1)
Reduce: sum counts per word

Step	What it does	Example
Map	Process input data into key–value pairs	(word, 1)
Shuffle	Groups all values by key	all counts for “cat” together
Reduce	Aggregates or summarizes per key	(“cat”, 25)

Run MapReduce Jobs using MRJOB

Java is the common language for running MapReduce jobs.

mrjob is a Python library developed by Yelp that lets us:

- Write MapReduce jobs entirely in Python.
- **mrjob** can run MaprReduce jobs locally, on Hadoop, or on EMR (AWS) with the same code.
- It allows us to focus on logic, not infrastructure.

Install mrjob on your local computer:

Open the command line terminal and type pip install mrjob

pip install mrjob

The simplest MRJob example: Word Count

We will write a simple MapReduce code and run it locally on our local machine.

Preparing Input Data

We'll use a simple text file for a word count example.
Open the command line terminal and type the following commands:

```
mkdir wordcount
cd wordcount
echo "hello world bye world hello hadoop
mapreduce world" > input.txt
```

The simplest MRJob example: Word Count

Create a python file, name it: wordcount.py

```
from mrjob.job import MRJob
class MRWordCount(MRJob):
    def mapper(self, _, line):
        for word in line.split():
            yield word, 1
    def reducer(self, word, counts):
        yield word, sum(counts)
```

Part	Description
`from mrjob.job import MRJob`	Imports the MRJob base class used to define MapReduce jobs.
`class MRWordCount(MRJob):`	Defines a new job called MRWordCount that inherits from MRJob.
`def mapper(self, _, line):`	The **mapper** function runs on each line of input text.
`for word in line.split():` `  yield word, 1`	Splits the line into words and emits each word paired with the number 1.
`def reducer(self, word, counts):`	The **reducer** function takes all counts for each word.
`yield word, sum(counts)`	Adds up all counts for a word and emits the total.

The simplest MRJob example: Word Count

Running the code locally:

```
python wordcount.py input.txt
```

You should see:

```
"bye"     1
"hadoop"  1
"hello"   2
"mapreduce" 1
"world"   3
```

This verifies that the logic works before distributing the job.

Running MRJob on Hadoop Using Docker

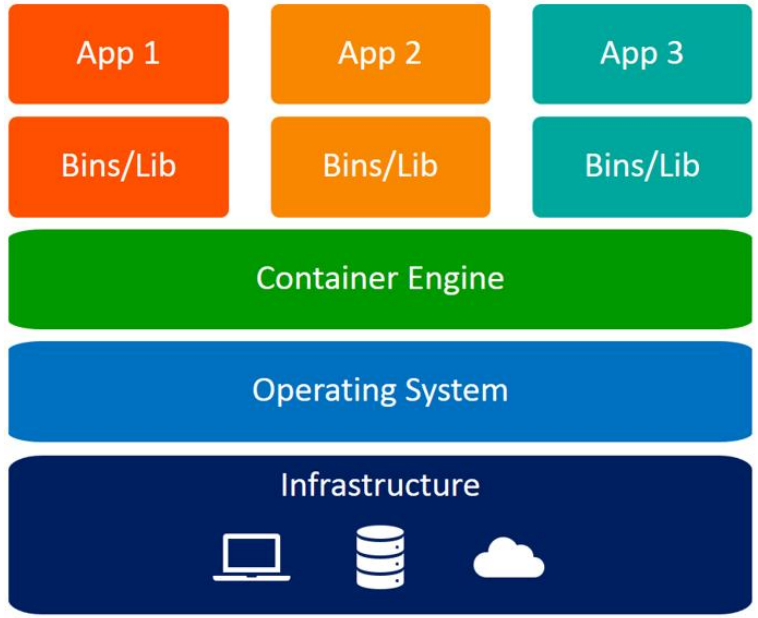
Introduction to Docker and Docker Compose

What is Docker

- Docker is a containerization platform that allows you to package an application together with all its dependencies into a single portable unit called a container.
- Containers ensure that the application runs the same way on any system, regardless of the underlying operating system or configuration.
- Unlike virtual machines, Docker containers share the host system's operating system kernel, making them lightweight, efficient, and fast to start.
- Analogy: A Docker container is like a sealed lab box that includes everything your program needs to run.

Why We Use Docker in This Lab

- To create a ready-to-use Hadoop environment without complex installation steps
- To prevent version or dependency conflicts between systems
- To make the lab easily reproducible on student laptops or classroom machines
- To simulate a cluster setup using a single machine



Containers

Installing and Configuring Docker

Step 1: Install Docker Desktop

1. Go to https://www.docker.com/products/docker-desktop
2. Download and install **Docker Desktop for Windows**.
3. During setup, **enable the WSL 2 backend**.
4. Restart your computer.
5. Verify Docker is running — the **whale icon** should appear in the system tray.

Step 2: Adjust Docker Resources

1. Open **Docker Desktop → Settings → Resources**.
2. Allocate:
   - CPUs: 2
   - Memory: 4–6 GB
   - Swap: 1–2 GB
3. Click **Apply & Restart**.

Proper resource allocation is critical. Hadoop requires enough memory for its daemons (NameNode, DataNode, ResourceManager, etc.) to start successfully.

Create a Docker Compose File

After installing docker, run the following command on the command line:

```
docker run -itd \
  --name hadoop \
  --hostname hadoop-master \
  -p 9870:9870 \
  -p 8088:8088 \
  sequenceiq/hadoop-docker:2.7.1 \
  /etc/bootstrap.sh -bash
```

Part	Meaning
docker run	Starts a new Docker container.
-itd	Runs it **interactive** (-i), with a **terminal** (-t), and in the **background** (-d).
--name hadoop	Names the container **hadoop** so it's easy to reference later.
--hostname hadoop-master	Sets the container's internal hostname to **hadoop-master**.
-p 9870:9870	Maps port **9870** (NameNode web UI) from container to host.
-p 8088:8088	Maps port **8088** (YARN ResourceManager web UI) from container to host.
sequenceiq/hadoop-docker:2.7.1	Uses this Hadoop Docker image (version 2.7.1).
/etc/bootstrap.sh -bash	Runs the startup script to initialize Hadoop, then keeps a Bash shell active.

Verify Hadoop

Now you are inside a Linux shell within the Hadoop environment.
Try:

```
hadoop version
```

→ Confirms Hadoop is installed.

```
hadoop fs -ls /
```

→ Lists directories in the Hadoop Distributed File System (HDFS).

Accessing Hadoop Interfaces

Interface	URL	Description
HDFS NameNode UI	http://localhost:9870	Lets you explore the distributed file system, upload files, and view storage metrics.
YARN ResourceManager UI	http://localhost:8088	Lets you track running and completed jobs, check logs, and monitor resources.

Useful Commands

Action	Command
Exit Hadoop shell	exit
Re-enter running container	docker exec -it hadoop bash
Check if it's running	docker ps
Start it again if stopped	docker start -ai hadoop

Preparing Input Data

We'll use a simple text file for a word count example.
Run the following commands on Windows Command line:

```
mkdir /wordcount
cd /wordcount
echo "hello world bye world hello hadoop mapreduce world" > input.txt
```

Upload this file to HDFS:

```
hadoop fs -mkdir /input
hadoop fs -put input.txt /input/
hadoop fs -ls /input
```

Uploading files into HDFS simulates the process of distributing data across multiple nodes.

Running on Hadoop

Run it through Hadoop Streaming:

```
python wordcount.py -r hadoop hdfs:///input/input.txt -o hdfs:///output_mrjob
```

You can watch progress on http://localhost:8088.

Option	Description
-r hadoop	Tells mrjob to use Hadoop's MapReduce engine.
hdfs:///input/input.txt	Input file path in HDFS.
-o hdfs:///output_mrjob	Output directory where Hadoop will store results.

Viewing the Results

After completion:

```
hadoop fs -cat /output_mrjob/part-00000
```

Expected output:

```
"bye"     1
"hadoop"  1
"hello"   2
"mapreduce" 1
"world"   3
```

Hadoop's reducer automatically aggregates counts and writes them to `part-00000`.

Understanding What Happened

- Hadoop divided your file into chunks (input splits).
- Each **mapper** processed one split, generating intermediate (word, 1) pairs.
- Hadoop grouped identical keys (all “hello”s together).
- The **reducer** summed values for each key.
- Results were stored in /output_mrjob as one file per reducer.
- This is the essence of **MapReduce** — distributed computation through key-value pair processing.

Key Takeaways

- Hadoop allows distributed processing using MapReduce.
- Docker provides an easy way to run Hadoop without complex installation.
- HDFS stores data across multiple nodes; YARN manages job execution.
- mrjob makes writing Python MapReduce programs straightforward.
- Hadoop web interfaces offer valuable visualization and debugging tools.

Optional Exercise

Try these for practice:

- Create a new input file (e.g., poem.txt).
- Upload it to HDFS.
- Run the same MapReduce script on it.
- Compare outputs with the first example.
- Observe job logs in YARN UI.

Movie Ratings Count Example

User ID| Movie ID| Rating | Time Stamp
0 50 5 881250949
0 172 5 881250949
0 133 1 881250949
196 242 3 881250949
186 302 3 891717742
22 377 1 878887116
244 51 2 880606923
166 346 1 886397596
298 474 4 884182806
115 265 2 881171488
253 465 5 891628467
305 451 3 886324817

What is MRStep in MRJob

- MRStep allows you to define **multiple MapReduce phases** (steps) inside a single MRJob program. Each step can have its own **mapper**, **combiner**, and **reducer**, enabling complex, multi-stage workflows.

Syntax Example

python Copy code

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

class MRWordFrequency(MRJob):
    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.reducer_sort_counts)
        ]
```

Simple Format (Single Step)

python Copy code

```python
class MRWordCount(MRJob):
    def mapper(self, _, line):
        for word in line.split():
            yield word, 1
    def reducer(self, word, counts):
        yield word, sum(counts)
```

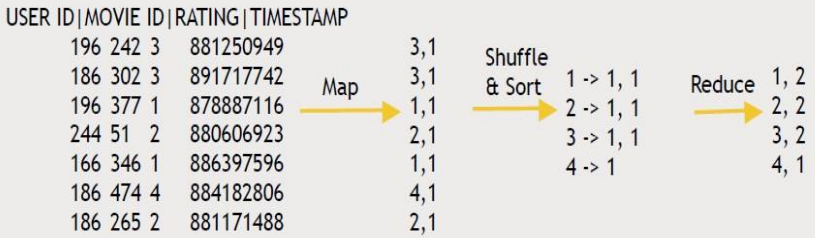
Comparison

Feature	Simple MRJob	MRStep
Number of steps	One	Multiple
Structure	Implicit (mapper/reducer only)	Explicit with `steps()`
Use case	Simple aggregation (e.g., word count)	Multi-phase jobs (e.g., sort, join, filter)
Flexibility	Limited	High – allows chaining operations

Map Reduce Movie Ratings Count Example

```
from mrjob.job import MRJob
from mrjob.step import MRStep
class RatingsBreakdown(MRJob):
    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_ratings,
                   reducer=self.reducer_count_ratings)
        ]
    def mapper_get_ratings(self, _, line):
        (userID, movieID, rating, timestamp) = line.split('\t')
        yield rating, 1
    def reducer_count_ratings(self, key, values):
        yield key, sum(values)
if __name__ == '__main__':
    RatingsBreakdown.run()
```

Writing the Mapper



Part	Description
`from mrjob.job import MRJob`	Imports the MRJob base class.
`from mrjob.step import MRStep`	Imports MRStep, used to define MapReduce steps.
`class RatingsBreakdown(MRJob):`	Defines a MapReduce job named `RatingsBreakdown`.
`def steps(self):`	Declares the sequence of steps in the job — here, just one step.
`MRStep(mapper=..., reducer=...)`	Specifies which functions act as the mapper and reducer.
`def mapper_get_ratings(self, _, line):`	Mapper function. Splits each line (tab-separated) into four values: `userID`, `movieID`, `rating`, and `timestamp`.
`yield rating, 1`	Emits the rating value as the key, with `1` as its count.
`def reducer_count_ratings(self, key, values):`	Reducer function. Receives each rating key and a list of counts.
`yield key, sum(values)`	Sums up how many times each rating occurred.
`if __name__ == '__main__': RatingsBreakdown.run()`	Standard Python entry point — runs the job when the script is executed.

Using MRStep (for multi-step jobs)

When you have multiple map/reduce phases, use MRStep.
For example, counting word frequency then sorting by frequency.

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

class MRWordFrequency(MRJob):
    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.reducer_sort_counts)
        ]

    def mapper_get_words(self, _, line):
        for word in line.split():
            yield word.lower(), 1

    def reducer_count_words(self, word, counts):
        yield None, (sum(counts), word)

    def reducer_sort_counts(self, _, word_count_pairs):
        for count, word in sorted(word_count_pairs,
reverse=True):
            yield word, count

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

Part	Description
`from mrjob.job import MRJob`	Imports MRJob, the base class for defining MapReduce jobs.
`from mrjob.step import MRStep`	Allows chaining multiple map-reduce steps.
`class MRWordFrequency(MRJob):`	Defines a custom MapReduce job.
`def steps(self):`	Defines a **two-step job**: first to count words, second to sort them.
Step 1: `mapper_get_words`	Reads each line, splits into words, converts to lowercase, and emits `(word, 1)`.
Step 1: `reducer_count_words`	Sums up counts per word and emits a tuple `(total_count, word)` under a single key (`None`) — this prepares for global sorting.
Step 2: `reducer_sort_counts`	Receives all `(count, word)` pairs, sorts them by count in descending order, and outputs `(word, count)`.
`if __name__ == '__main__': MRWordFrequency.run()`	Standard entry point — runs the job.