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:**

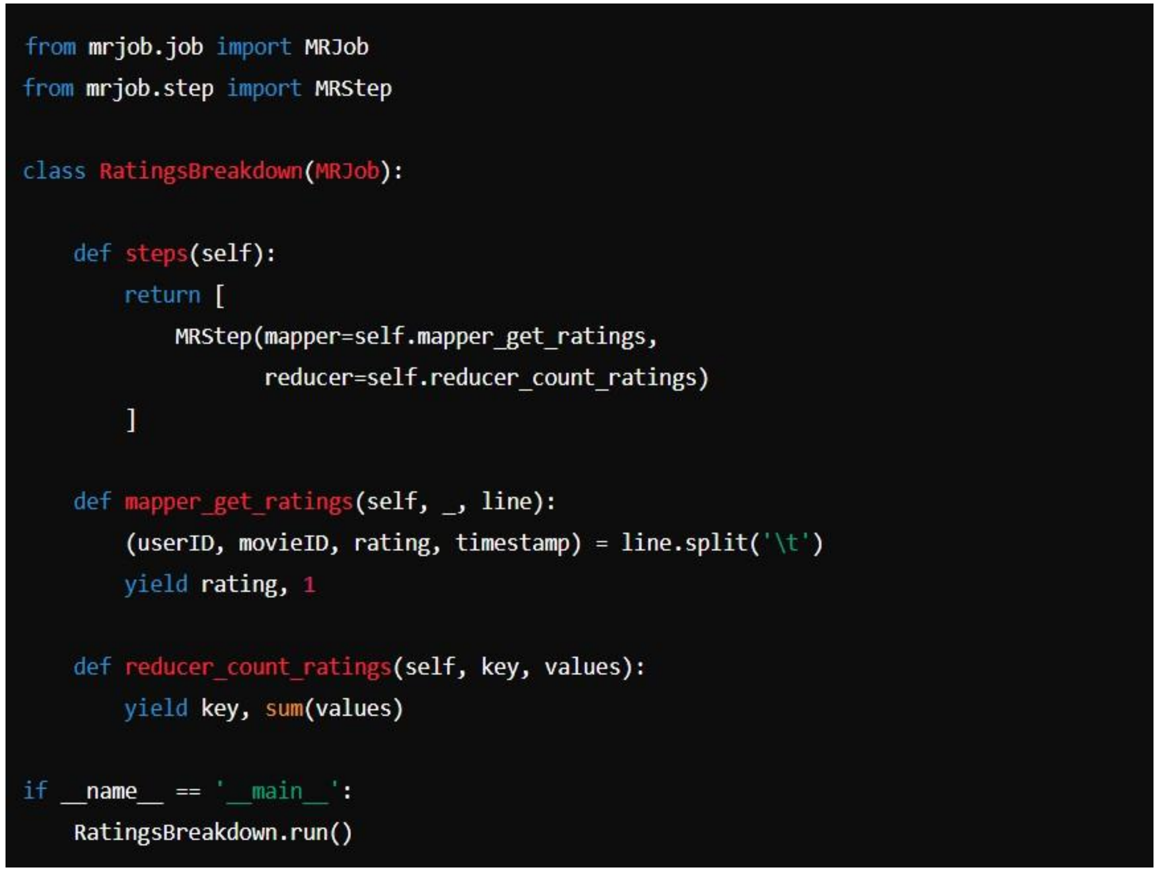
```
def add_reduce(x, y):
  out = x + y
  print(f"{x}+{y}-->{out}")
  return out
reduce(add_reduce,lst)

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

Map Reduce Using Hadoop and MRJob Python Library

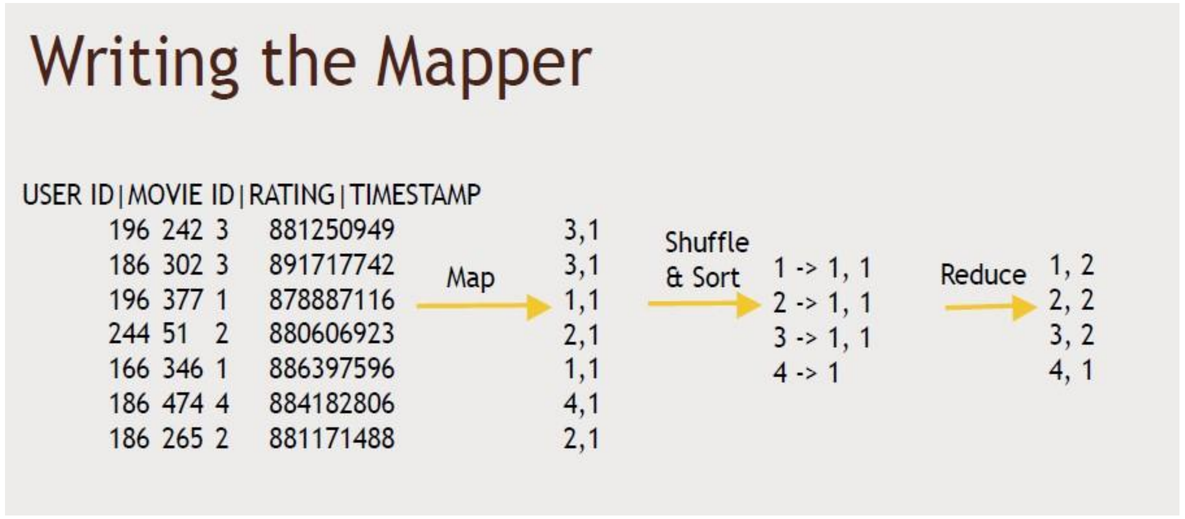
Map Reduce Movie Ratings Count Example

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

Map Reduce Movie Ratings Count Example



```python
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

```
USER ID|MOVIE ID|RATING|TIMESTAMP
      196  242  3    881250949
      186  302  3    891717742
      196  377  1    878887116
      244  51   2    880606923
      166  346  1    886397596
      186  474  4    884182806
      186  265  2    881171488
```

One Step MapReduce 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()
```

Two Steps MapReduce (Sorting after Counting)

```
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),
            MRStep(reducer=self.reducer_sorted_output)
        ]

    def mapper_get_ratings(self, _, line):
        (userID, movieID, rating, timestamp) = line.split('\t')
        yield movieID, 1

    def reducer_count_ratings(self, key, values):
        yield str(sum(values)).zfill(5), key

    def reducer_sorted_output(self, count, movies):
        for movie in movies:
            yield movie, count
if __name__ == '__main__':
    RatingsBreakdown.run()
```

BUSINESS INTELLIGENCE AND DATA ANALYTICS
– WE PREDICT THE FUTURE –

Institute of Analytics
Recognized Programme

University of Petra

Running 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

What is Docker Compose

Docker Compose is a tool used to define and manage multi-container applications.
It uses a configuration file named docker-compose.yml that describes which containers to start, what images to use, which ports to expose, and how services interact.
Once defined, the entire environment can be launched with one command:

```
docker-compose up -d
```

Why Docker Compose for Hadoop

- Hadoop consists of several components (NameNode, DataNode, ResourceManager, etc.).
  Docker Compose allows all of these to be started and configured together as one environment.
  In this lab, a single Compose file runs a single-node Hadoop cluster inside one container for simplicity.

Key Takeaways

- Docker provides isolated, consistent environments for running applications.
- Docker Compose automates the setup and coordination of multiple containers or services.
- Together, they allow us to run Hadoop and its ecosystem easily, reproducibly, and without installation issues.

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

Create a file named `docker-compose.yml` and add:

```
version: '3'
services:
  hadoop:
    image: sequenceiq/hadoop-docker:2.7.1
    container_name: hadoop
    hostname: hadoop-master
    ports:
      - "9870:9870"
      - "8088:8088"
    command: /etc/bootstrap.sh -bash
    tty: true
```

The YAML file tells Docker which Hadoop image to use and how to expose ports for the Hadoop web interfaces.

Start Hadoop

Start Hadoop

```
docker-compose up -d
```

Then verify:

```
docker ps
```

You should see the container named **hadoop** running.
This single container simulates a full Hadoop cluster.

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.

Accessing the Hadoop Shell

Open a terminal inside the running container:

```
docker exec -it hadoop /bin/bash
```

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).

Preparing Input Data

We’ll use a simple text file for a word count example.

```
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.

Installing Python and mrjob

Hadoop natively runs Java MapReduce, but we can use **Python** with the `mrjob` library for easier experimentation. Install tools inside the container:

```
apt-get update
apt-get install -y python3-pip
pip3 install mrjob
```

`mrjob` handles communication between your Python script and Hadoop's streaming interface.

Writing a Python MapReduce Script

Writing a Python MapReduce Script

Inside `/wordcount`, create `wordcount_mrjob.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)
```

Here, you define:

- one `mapper()` method
- one `reducer()` method

`mrjob` automatically assumes **a single MapReduce phase**, so you don't need to use `MRStep`.

This is perfect for small tasks like word counting.

Explanation:

•The **mapper()** reads each line, splits it into words, and emits each word as a key with value `1`.

•The **reducer()** collects all the values (counts) for each word and sums them.

•Hadoop automatically handles shuffling and sorting between mapper and reducer.

Testing Locally

Before running on Hadoop, test it locally:

```
python3 wordcount_mrjob.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 on Hadoop

Run it through Hadoop Streaming:

```
python3 wordcount_mrjob.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`.

Visualizing via Web Interfaces

a. HDFS NameNode UI

Open http://localhost:9870

Browse to /input and /output_mrjob

You can open, download, or upload files

b. YARN ResourceManager UI

Open http://localhost:8088

Observe your job’s status, execution time, and logs

These dashboards make it easier to explain Hadoop’s parallel job handling during class.

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.

Troubleshooting

Problem	Likely Cause	Solution
Docker daemon not running	Docker service stopped	Start Docker Desktop
hadoop-streaming*.jar not found	Container not fully initialized	Restart the container
Permission denied on script	Missing execute permission	chmod +x script.py
Output directory exists	Hadoop prevents overwriting	hadoop fs -rm -r /output_mrjob
Memory errors	Insufficient allocation	Increase Docker memory to 6 GB

Stopping and Cleaning Up

When done:

```
exit
docker-compose down
```

This stops the Hadoop container and releases system resources.

Optional Student 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.

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.