



**POLYTECHNIQUE  
MONTRÉAL**

**UNIVERSITÉ  
D'INGÉNIERIE**

LOG8415E – Advanced Concepts of Cloud Computing

Assignment #2

**MapReduce on AWS**

Alexis Desforges - 2146454

Samy Labassi - 1953898

Louis Lalonde - 2335476

Loic Nguemegne Temena - 2180845

Submitted to

Vahid Majdinasab

Fall 2025 – 4th November 2025

# Contents

<b>1</b>	<b>WordCount Experiments</b>	<b>2</b>
1.0.1	Experiments with WordCount program . . . . .	2
1.0.2	Performance comparison of Hadoop vs. Linux . . . . .	3
1.0.3	Performance comparison of Hadoop vs. Spark . . . . .	4
<b>2</b>	<b>MapReduce on AWS</b>	<b>6</b>
2.0.1	Overview of MapReduce and the social network problem . . . . .	6
2.0.2	Algorithm implementation . . . . .	7
2.0.3	MapReduce implementation . . . . .	7
2.0.4	Results recommendation . . . . .	11
<b>3</b>	<b>Instructions to run the experiment code</b>	<b>12</b>

# Chapter 1

## WordCount Experiments

### 1.0.1 Experiments with WordCount program

MapReduce is a classical paradigm in big data processing. It consists of two main stages: a *map* phase that transforms the input data, and a *reduce* phase that aggregates or summarizes the results [1]. The *WordCount* program is often considered the “Hello World” of big data, as it illustrates the basic workflow of a MapReduce job. The goal of this program is to transform an input text into a list of words (assuming words are separated by spaces) and then count the occurrences of each word.

In our experiment, we implemented the WordCount task in three different ways: using Bash, Hadoop, and Spark. This allowed us to compare the strengths and weaknesses of the MapReduce paradigm across different technologies.

**1. WordCount using Bash.** In Linux, we can simulate a MapReduce operation using standard shell commands. The following command performs a word count using basic Bash tools:  
`cat {file_path} | tr ' ' '\n' | sort | uniq -c | sort -nr > output/linux_{file_name}_output.txt`

Here, the *map* phase is implemented by `cat {file_path} | tr ' ' '\n' | sort` Which reads the file, replaces spaces with newline characters (thus transforming the text into a list of words), and sorts them alphabetically. The *reduce* phase is performed by `uniq -c | sort -nr`, which counts identical words (using `-c`) and sorts the results by decreasing frequency. This method is simple and efficient for small files, but it does not scale well to large datasets or distributed environments.

**2. WordCount using Hadoop.** Hadoop provides a framework to execute MapReduce jobs in a distributed manner. Before execution, the input file must be stored in the Hadoop Distributed File System (HDFS) so that it is accessible to all cluster nodes. Hadoop automatically splits the input file into blocks (called *splits*), which are distributed among worker nodes. Each node independently executes the map and reduce phases on its assigned data, and the results are then merged to form the final output.

The WordCount job in Hadoop can be launched with the following command:

```
hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/  
hadoop-mapreduce-examples-3.4.2.jar wordcount {file_path} output
```

This command runs the built-in Hadoop example for WordCount. The framework handles input distribution, intermediate data shuffling, and fault tolerance. Although Hadoop offers robustness and scalability, it relies heavily on disk I/O between stages, which can limit performance for iterative or in-memory workloads.

**3. WordCount using Apache Spark.** Apache Spark is a more modern big data framework that builds upon the Hadoop ecosystem but emphasizes in-memory computation. Unlike Hadoop, which writes intermediate results to disk, Spark keeps data in memory using *Resilient Distributed Datasets* (RDDs), significantly improving performance for iterative and interactive workloads.

The WordCount example in Spark can be executed as follows:

```
spark-submit --master local[2] --class org.apache.spark.examples.JavaWordCount \
/usr/local/spark/examples/jars/spark-examples_2.13-4.0.1.jar {file_path} output
```

In this command, the job is executed locally using two threads (`--master local[2]`). The `JavaWordCount` class defines the MapReduce logic using RDD transformations such as `flatMap`, `mapToPair`, and `reduceByKey`. Spark automatically handles resource allocation, task scheduling, and fault tolerance, while keeping intermediate data in memory whenever possible. This makes Spark faster than Hadoop for most workloads, especially those involving repeated computations or data reuse.

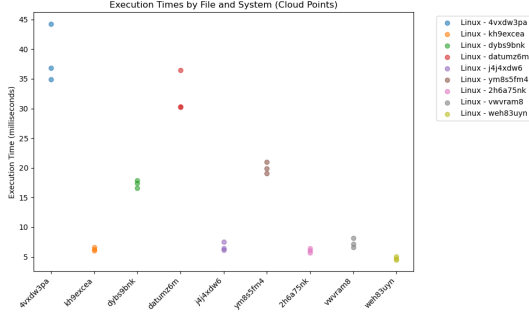
**Methodology.** These three implementations highlight different aspects of the MapReduce paradigm. The Bash version is simple but limited to single-machine processing. Hadoop scales to large clusters but incurs significant disk I/O overhead. Spark, by contrast, combines scalability with in-memory processing, providing better performance and flexibility for modern big data analytics. Let's now analyze the performance results of these implementations on various input files. To compare these three MapReduce paradigms, we used nine different input files of varying sizes and contents. Each algorithm (Bash, Hadoop, and Spark) was executed three times on every file to minimize the effect of random fluctuations and system noise. The average execution time for each method was then computed.

To better visualize the results, we plotted two types of diagrams: a scatter plot and a histogram. The scatter plot illustrates the relationship between input file size and execution time, highlighting how each framework scales with larger datasets. The histogram, on the other hand, compares the average execution time of the three approaches, allowing a clear visualization of their relative performance.

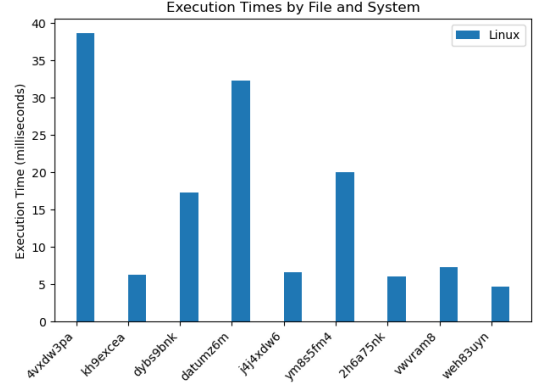
Through this experimental setup, we aim to quantify the trade-offs between simplicity, scalability, and processing speed across Bash, Hadoop, and Spark implementations of the WordCount program.

### 1.0.2 Performance comparison of Hadoop vs. Linux

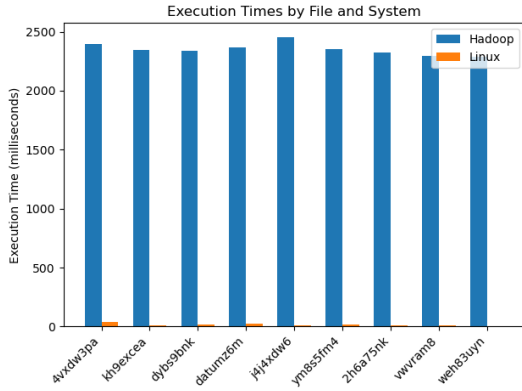
During our experiments comparing Hadoop and Linux environments, we executed the same MapReduce algorithm three times on a set of nine files. The execution results were recorded and visualized in the four images below, which illustrate performance differences through cloud point distributions and histogram comparisons.



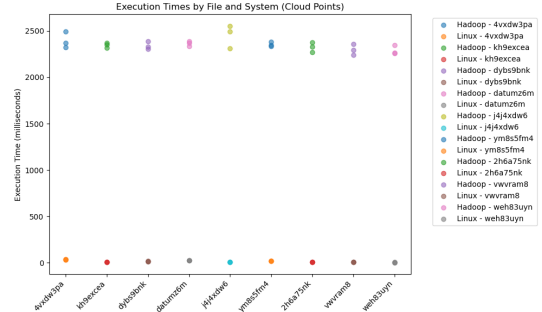
(a) Linux (only) cloud points



(b) Linux (only) histogram



(c) Hadoop and Linux histogram



(d) Hadoop and Linux cloud points

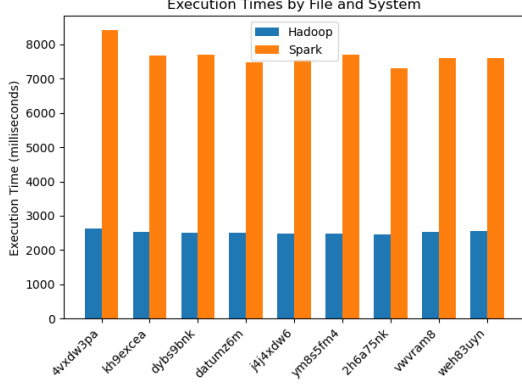
Figure 1.1: Execution time visualizations for Linux and Hadoop

The results illustrated in figure 1.1 provide a comparative analysis of execution times for the WordCount program across Linux (Bash) and Hadoop implementations. Figure (a) presents the scatter plot of execution times for the Linux-only implementation, while Figure (b) shows the corresponding histogram of average execution times. Figures (c) and (d) extend the comparison by including Hadoop results Figure (c) displaying the mean execution times for both systems, and Figure (d) the combined scatter plot of all runs.

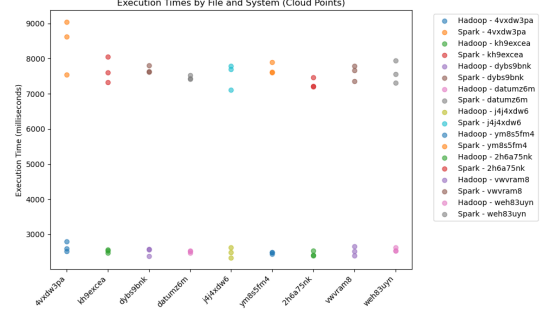
From these visualizations, we observe that the Linux (Bash) implementation completes the WordCount task in an average of approximately 14 milliseconds, whereas Hadoop requires around 2500 milliseconds. This significant difference is explained by Hadoop's inherent overhead: before processing, it must initialize multiple nodes, distribute data blocks, and coordinate the execution of map and reduce tasks across the cluster. While this overhead makes Hadoop less efficient for small to medium-sized datasets such as those used in our experiment it becomes advantageous for large-scale distributed data processing, where its parallelism and fault tolerance outweigh the initialization costs. In summary, Bash performs best for lightweight, local processing, whereas Hadoop is designed for scalability across massive datasets.

### 1.0.3 Performance comparison of Hadoop vs. Spark

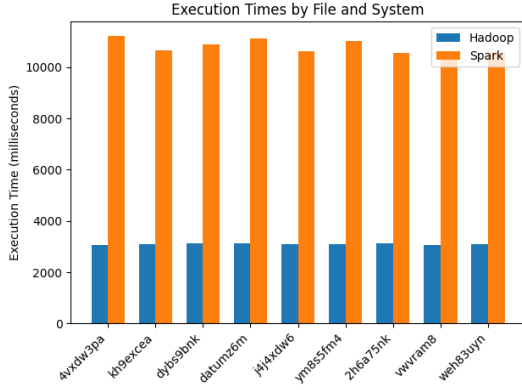
We repeated the same experiment, this time comparing Hadoop and Spark implementations on AWS infrastructure and on local machines. The execution results are visualized in the four images below, which illustrate performance



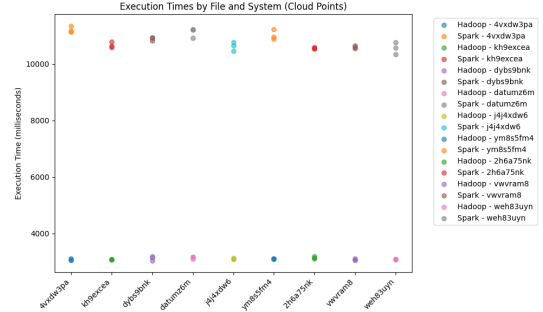
(a) Hadoop and Spark histogram on local machine



(b) Hadoop and Spark cloud points on local machine



(c) Hadoop and Spark histogram on EC2 instance



(d) Hadoop and Spark cloud points on EC2 instance

Figure 1.2: Execution time visualizations for Hadoop and Spark

The results illustrated in figure 1.2 provide a comparative analysis of execution times for the WordCount program across Hadoop and Spark implementations on AWS and on a local machine. Although Spark is generally known for its speed in distributed in-memory processing, our experimental results show that it performs slower than Hadoop for the WordCount problem on relatively small input files.

As shown in the diagrams, Spark takes on average around 7500 milliseconds to complete the task locally, compared to approximately 2500 milliseconds for Hadoop. This difference can be explained by the initialization overhead of the Spark engine, its lazy evaluation model, and internal optimizations that are not effectively leveraged for such a simple workload. Spark introduces additional latency due to its complex task scheduling and execution orchestration, which are designed for large-scale data pipelines involving iterative or interactive computations. We can also see that running the same experiment on AWS EC2 instances yields slower results for Spark, likely because the EC2 VM (t2.large) has lower single-core/turbo performance and higher I/O/virtualization overhead (EBS-backed disk, shared CPU) than the local machine that was used, Spark's JVM startup and executor scheduling incur extra latency of about 2 seconds.

In contrast, Hadoop executes the MapReduce paradigm in a more direct and sequential manner, making it more efficient for small, straightforward jobs such as WordCount. Its execution time on the EC2 VM is also very similar to that on the local machine, further emphasizing its suitability for this type of workload. Therefore, while Spark generally excels in large-scale, iterative, or memory-intensive processing, Hadoop proves to be faster and more appropriate for this specific benchmark and dataset size.

## Chapter 2

# MapReduce on AWS

### 2.0.1 Overview of MapReduce and the social network problem

The general idea behind the social network friend recommendation algorithm is to leverage mutual friends as indicators to suggest new connections. Each node in the network acts as a common friend bridging its directly connected neighbors. To avoid recommending users who are already friends, the edges linking a node to its friends serve to invalidate the mutual friend count between those two nodes. This is crucial since a node may not appear in every computation; hence, ensuring no current friends are recommended is necessary.

Without MapReduce, the algorithm begins by iterating through every node to identify its direct friends and explicitly mark those pairs with a negative infinity value to exclude them from future mutual friend computations. Following that, for each ordered combination of the node's friends (excluding permutations), the algorithm increments a count for the pair, representing that they share a common friend (the node being processed). In the final step, the algorithm amalgamates counts from both sides of every pair to gather all potential friend recommendations for each node, then sorts these identified recommendations in descending order based on the mutual friend count.

This problem aligns exceptionally well with the MapReduce paradigm since the first phase relies only on local information involving a node and its friends, enabling parallel processing across multiple mapper instances. In the Map step, the algorithm emits each friend pair as a key with a value of one, denoting a single mutual friend. Simultaneously, it emits entries marking pairs of already connected friends with a negative indicator to invalidate recommendations for existing friendships. The Reduce step aggregates these counts for each friend pair, representing the total number of mutual friends they share. Lastly, the reducer outputs the user as the key and the recommended friend along with the mutual friend count as the value for both nodes in the pair. A third aggregation step then collects all recommendations for each node, sorts them by mutual friend counts, and formats the output accordingly.

Below is LaTeX code for the pseudocode representing both the basic algorithm and its MapReduce adaptation:

## 2.0.2 Algorithm implementation

---

**Algorithm 1** Basic Friend Recommendation Algorithm

---

```
1: for each node  $u$  do
2:   Let  $F_u$  be the friends of  $u$ 
3:   for each friend  $f$  in  $F_u$  do
4:     Invalidate recommendation for pair  $(u, f)$  by setting score to  $-\infty$ 
5:   end for
6:   for each unordered pair  $(f_i, f_j)$  in  $F_u$  do
7:     Increment mutual friend count for pair  $(f_i, f_j)$  by 1
8:   end for
9: end for
10: Aggregate counts for each pair to get mutual friend counts
11: For each node, gather recommended friends sorted by mutual friend count
```

---

---

**Algorithm 2** MapReduce Friend Recommendation Algorithm

---

```
1: procedure MAPPER
2:   Input: node  $u$  and its friend list  $F_u$ 
3:   for each friend  $f$  in  $F_u$  do
4:     Emit key= $(u, f)$ , value= $-\infty$  ▷ Invalidate direct friends
5:   end for
6:   for each unordered friend pair  $(f_i, f_j)$  in  $F_u$  do
7:     Emit key= $(f_i, f_j)$ , value=1 ▷ One mutual friend  $u$ 
8:   end for
9: end procedure
10: procedure REDUCER
11:   Input: key=pair  $(a, b)$ , values=list of counts or  $-\infty$ 
12:   Sum values ignoring  $-\infty$  unless present, in which case mark pair as direct friends
13:   if pair is not direct friends then
14:     Emit  $(a, b, \text{mutual friend count})$  and  $(b, a, \text{mutual friend count})$ 
15:   end if
16: end procedure
17: procedure FINAL AGGREGATION
18:   for each node  $u$  do
19:     Collect all  $(u, r, \text{count})$  pairs
20:     Sort recommended friends  $r$  by descending count and ascending user ID
21:     Output recommendations for  $u$ 
22:   end for
23: end procedure
```

---

The friendship recommendation algorithm presented in this report is based on the approach discussed in the Stack Overflow post titled 'Hadoop M/R to implement People You Might Know friendship recommendation'. This post outlines how mutual friends can be leveraged to suggest new connections in a social network using the MapReduce framework.

## 2.0.3 MapReduce implementation

Our implementation is modular: each MapReduce function (orchestrator, mapper, partitioner, reducer) is a small FastAPI web service and the deployment helper provisions EC2 instances,

uploads the code and starts the services. In figure 2.1, we depict the architecture used for our MapReduce AWS implementation for the social network problem experiment.

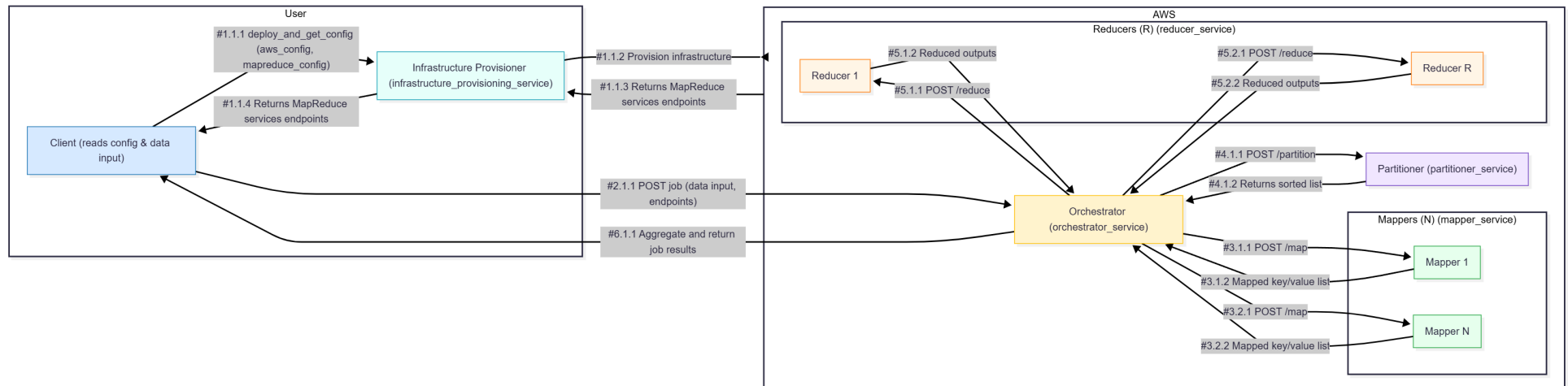


Figure 2.1: AWS MapReduce architecture and data flows on AWS.

## Components and responsibilities

- Client: reads configuration, calls the infrastructure provisioner, and submits a job to the orchestrator. See `mapreduce/mapreduce_client.py` (`mapreduce/mapreduce_client.py`).
- Infrastructure provisioner: provisions EC2 instances, uploads the service files, starts each service and waits for `/health` endpoints. The helper is implemented in `mapreduce/infrastructure_provisioning.py` (`mapreduce/infrastructure_provisioning.py`) and exposes `infrastructure_provisioning.deploy_and_get_config` (see `mapreduce/infrastructure_provisioning.py`).
- Orchestrator: receives the job payload from the client, loads the data file, splits data input lines across mappers, collects mapper outputs, sends sorted data to the partitioner, partitions data based on an MD5 hash of data keys, distributes partitions to reducers, gathers reduced outputs, performs a final aggregation step and saves the final result. See the orchestrator implementation in `mapreduce/orchestrator_service.py` (`mapreduce/orchestrator_service.py`) and the main MapReduce driver function `orchestrator_service.run_mapreduce_job` (see `mapreduce/orchestrator_service.py`).
- Mapper: exposes a POST `/map` endpoint that loads a user algorithm module at runtime and runs its `map_function` over assigned input lines. See `mapreduce/mapper_service.py` (`mapreduce/mapper_service.py`) and the mapper handler `mapper_service.map_data` (see `mapreduce/mapper_service.py`).
- Partitioner: takes sorted mapper output and splits it deterministically into N partitions using a key-hash function (MD5) so identical keys go to the same reducer and the data keys are roughly uniformly distributed. See `mapreduce/partitioner_service.py` (`mapreduce/partitioner_service.py`) and the partitioner handler `partitioner_service.partition_data` (see `mapreduce/partitioner_service.py`).
- Reducer: loads the same algorithm module and executes its `reduce_function` on each group of values for a key. See `mapreduce/reducer_service.py` (`mapreduce/reducer_service.py`) and the reducer handler `reducer_service.reduce_data` (see `mapreduce/reducer_service.py`).

## Dataflow

1. Client posts a job to the orchestrator with `input_file` (path to the data input file) and runtime endpoints (mappers, reducers, partitioner). See the client request flow in `mapreduce/mapreduce_client.py`.
2. Orchestrator executes the pipeline implemented in `orchestrator_service.run_mapreduce_job` (see `mapreduce/orchestrator_service.py`):
  - Split input lines into mapper chunks and POST each chunk to a mapper's `/map` endpoint.
  - Collect all mapped key/value emissions from mappers and sort them by key.
  - POST the sorted list to the partitioner `/partition` endpoint to obtain partitions.
  - For each partition, POST to a reducer's `/reduce` endpoint and gather reduced outputs.
  - Optionally call the algorithm's aggregate function (loaded dynamically)
  - Persist the final MapReduce job output to the `./output` directory.
3. Each service exposes a `/health` endpoint used by the deployer to confirm readiness.

**Pluggable algorithms** Algorithms are standard Python modules placed under `mapreduce/algorithms/`. The orchestrator and services dynamically load a requested algorithm file (see `load_algorithm` in `mapreduce/orchestrator_service.py` and the similar loaders in `mapreduce/mapper_service.py` and `mapreduce/reducer_service.py`). Example: the Friend recommendation algorithm is in `mapreduce/algorithms/friendrec.py` and follows the required `map_function / reduce_function` contract.

**Notes on scaling and determinism** The partitioner uses a stable hash on the record key to ensure that identical keys are consistently routed to the same reducer (see the key-hash logic in `mapreduce/partitioner_service.py`). The orchestrator can scale the number of mappers and reducers by configuration (see `mapreduce/configs/mapreduce_config.json` used by the deployer in `mapreduce/infrastructure_provisioning.py`).

**Limitations and improvements** Currently, after the mappers process their assigned data chunks, all mapper results are sent back to the orchestrator service for aggregation. If the dataset is large and the map operation produces a significant increase in the number of output records, this can create a bottleneck and performance issues at the orchestrator level, since it must aggregate and forward all mapper outputs to the partitioner and reducers.

An effective improvement is to implement map-side partitioning, where each mapper computes the partition ID for each output record using the partitioner hash function. Instead of sending all mapper outputs to the orchestrator, each mapper batches and sends its partitioned data directly to the corresponding reducer. This approach reduces the load on the orchestrator, distributes network traffic more evenly, and enables better parallelism.

## 2.0.4 Results recommendation

Here are the recommended users for each target user, along with the amount of common friends:

- 9019: (9022, 2), (317, 1), (9023, 1)
- 9020: (9021, 3), (9016, 2), (9022, 2), (9017, 2), (317, 1), (9023, 1)
- 9021: (9020, 3), (9016, 2), (9017, 2), (9022, 2), (317, 1), (9023, 1)
- 9022: (9020, 2), (9019, 2), (9021, 2), (317, 1), (9016, 1), (9017, 1), (9023, 1)
- 9990: (34485, 1), (37941, 1), (13478, 1), (13134, 1), (13877, 1), (34299, 1), (34642, 1)
- 9993: (9991, 5), (34485, 1), (37941, 1), (13478, 1), (13134, 1), (13877, 1), (34299, 1), (34642, 1)
- 924: (45881, 1), (15416, 1), (43748, 1), (439, 1), (11860, 1), (2409, 1), (6995, 1)
- 8941: (8943, 2), (8944, 2), (8940, 1)
- 9992: (9989, 4), (9987, 4), (35667, 3), (9991, 2)
- 8942: (8939, 3), (8940, 1), (8943, 1), (8944, 1)

## Chapter 3

# Instructions to run the experiment code

Instructions to run the project code are provided in the project `README.md` file. Please refer to the `README.md` for detailed setup and execution steps.

The project repository can be found at the following link (authorization required):  
<https://github.com/LouisLalonde/LOG8415E-TP2>

# Bibliography

- [1] Jim Holdsworth and Matthew Kosinski. What is MapReduce? | IBM, November 2024.