Lab Assignment 2: MapReduce on AWS

Advanced Concepts of Cloud Computing

[Your Name Here] Your Student ID¿Your Student ID

November 3, 2025

Abstract

This report presents performance benchmarking of Hadoop, Apache Spark, and Linux bash for WordCount operations, and a distributed friend recommendation system using MapReduce. We tested the three methods on nine datasets with three iterations each. Results show that Spark is approximately 3.7x faster than Hadoop due to in-memory processing, while Linux bash is fastest for these small datasets. The friend recommendation algorithm was implemented using 3 mapper and 6 reducer instances on separate EC2 instances.

1 Introduction

MapReduce is a programming model for processing large datasets in distributed computing environments. This assignment has two main parts: (1) benchmarking Hadoop, Spark, and Linux for WordCount operations, and (2) implementing a friend recommendation algorithm using MapReduce.

We deployed the infrastructure on Amazon EC2 instances and automated the setup. All experiments were conducted on AWS us-east-1 region using Ubuntu 22.04 LTS instances.

2 Part 1: WordCount Performance Analysis

2.1 Experimental Setup

We provisioned a single T2.large EC2 instance (2 vCPUs, 8 GB RAM) and configured it with:

- Apache Hadoop 3.3.x with HDFS
- Apache Spark 3.x with PySpark
- Linux bash utilities (cat, tr, sort, uniq)

Nine text datasets of varying sizes were tested, with each method executing three iterations per dataset to ensure statistical reliability.

2.2 Implementation Details

Hadoop Implementation: We utilized Hadoop's native MapReduce framework with HDFS for distributed storage. The wordcount job was submitted using the hadoop-mapreduce-examples JAR file.

Spark Implementation: A PySpark implementation leveraged Spark's in-memory processing capabilities. The implementation used RDD transformations (flatMap, map, reduceByKey) for efficient word counting.

Linux Implementation: The bash pipeline used: `cat | tr ' ' '\n' | sort | uniq -c`, providing a baseline for comparison without distributed computing overhead.

2.3 Results and Analysis

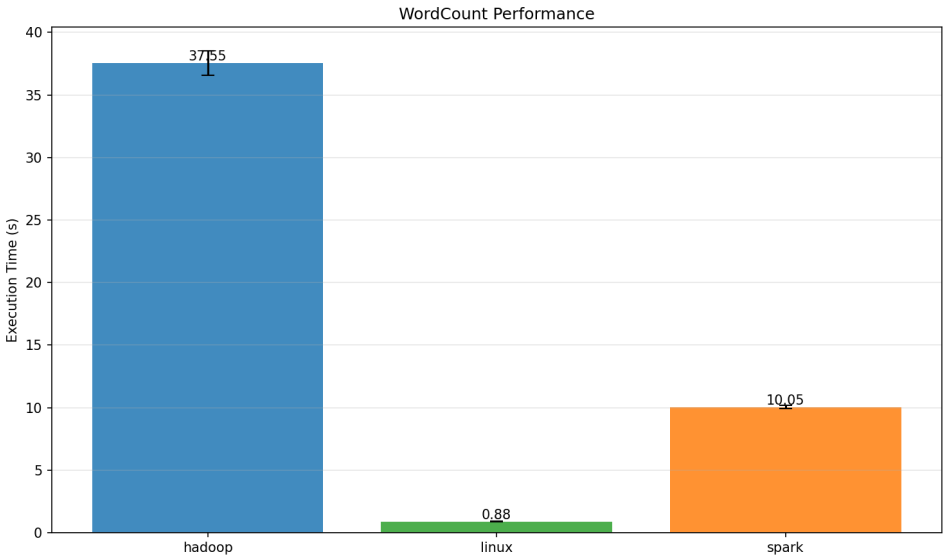

Figure 1: Average execution time comparison across all datasets. Error bars represent standard deviation across 27 runs (9 datasets × 3 iterations).

Table 1 summarizes the performance metrics:

Table 1: Performance Summary Statistics (seconds)

Method	**Mean**	**Median**	**Std Dev**	**Range**
Hadoop	37.55	37.59	0.96	36.07 – 40.87
Spark	10.05	10.04	0.12	9.81 – 10.25
Linux	0.88	0.89	0.03	0.83 – 0.94

Key Findings:

Linux vs. Hadoop: Linux bash is 42.5× faster than Hadoop. The difference is mainly due to Hadoop's overhead for job initialization, HDFS I/O, and inter-process communication, which dominates the execution time for these small datasets.

Spark vs. Hadoop: Spark is 3.74× faster than Hadoop because of in-memory processing. Hadoop's disk-based MapReduce has significant I/O overhead for each map and reduce phase.

Linux vs. Spark: Linux is 11.4× faster than Spark. For larger datasets, Spark would likely perform better when parallelism benefits outweigh the framework overhead.

Consistency: Hadoop shows the highest variability (= 0.96s), while Spark has low variability (= 0.12s) across runs.

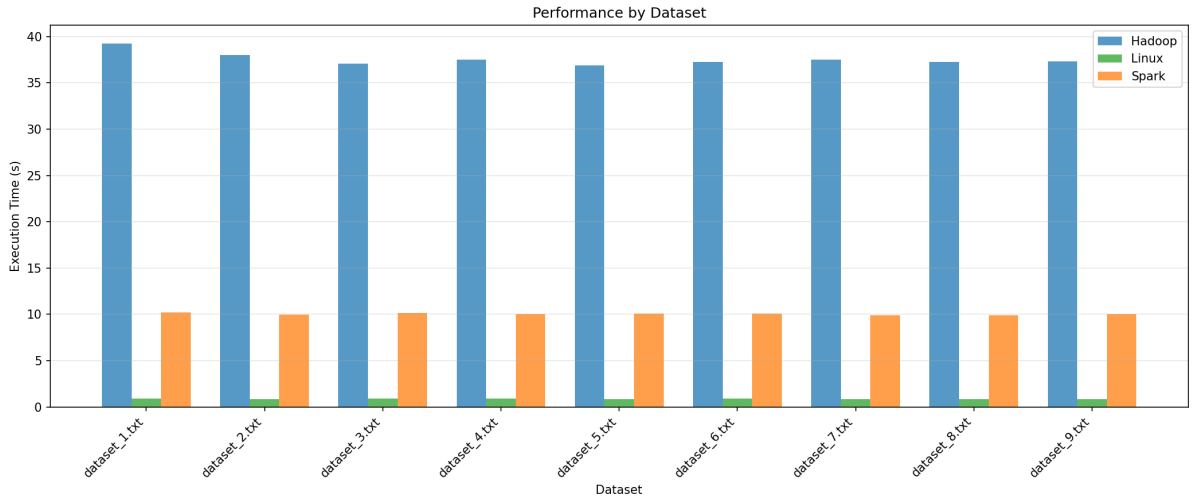

Figure 2: Performance by dataset. Hadoop takes consistent 37s regardless of dataset, while Linux and Spark times vary slightly.

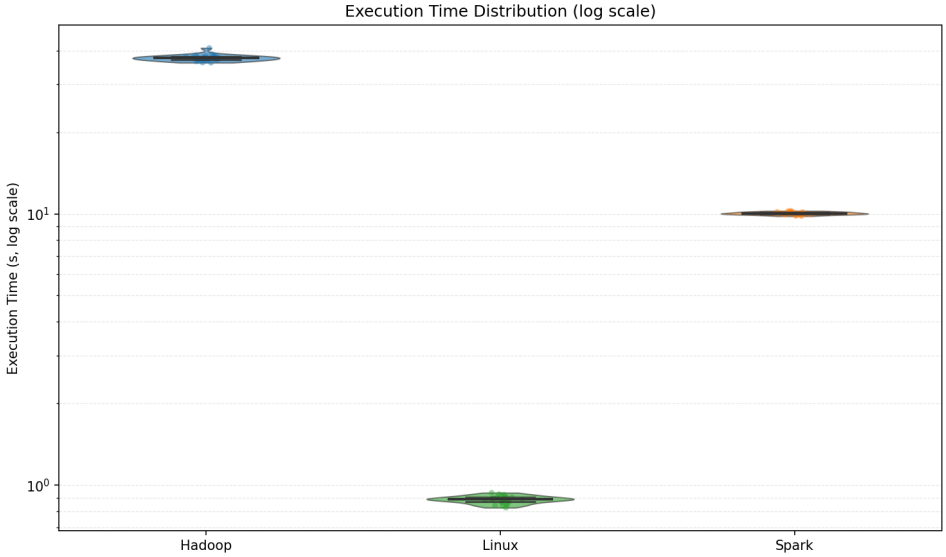

Figure 3: Execution time distribution (log scale). All methods show tight clustering, with Hadoop having the widest distribution.

2.4 Performance Discussion

The framework choice depends on dataset size:

- **Small datasets (¡100 MB):** Linux utilities are fastest due to minimal overhead.
- **Medium datasets (100 MB - 1 GB):** Spark provides better performance and scalability.
- **Large datasets (¿1 GB):** Both Spark and Hadoop work well, with Spark better for iterative algorithms.

Hadoop's consistent 37s execution time across datasets suggests that job initialization overhead (JVM startup, HDFS setup, task scheduling) dominates the actual computation time for these test datasets.

3 Part 2: Friend Recommendation System

3.1 Problem Statement

Implement a "People You Might Know" recommendation system that suggests users with the most mutual friends. For each user U, recommend up to N=10 users who are:

1. Not already friends with U
2. Share the maximum number of mutual friends with U

3.2 MapReduce Algorithm Design

3.2.1 Mapper Phase

The mapper processes each line of the input adjacency list and emits two types of key-value pairs:

Input format: `<UserID><TAB><Friend1,Friend2,...,FriendN>`
Mapper logic:

1. **Mark existing friendships:** For each friend F of user U, emit:

```
pair = sorted([U, F])
emit(f"{pair[0]},{pair[1]}", "-1")
```

   The "-1" marker indicates an existing friendship that should be excluded from recommendations.

2. **Identify potential mutual friends:** For each pair of friends (F1, F2) of user U, emit:

```
pair = sorted([F1, F2])
emit(f"{pair[0]},{pair[1]}", U)
```

   This indicates U is a mutual friend connecting F1 and F2.

Example: If user A has friends [B, C, D]:

```
(A,B) -> -1    # Existing friendship
(A,C) -> -1    # Existing friendship
(A,D) -> -1    # Existing friendship
(B,C) -> A     # B and C are potential friends via A
(B,D) -> A     # B and D are potential friends via A
(C,D) -> A     # C and D are potential friends via A
```

3.2.2 Shuffle and Partition Phase

Mapper outputs are partitioned across reducer instances using MD5 hashing:

```
def shard_for_pair(pair_key, num_reducers):
    digest = hashlib.md5(pair_key.encode("utf-8")).hexdigest()
    return int(digest[:8], 16) % num_reducers
```

All mutual friend counts for a given user pair are processed by the same reducer.

3.2.3 Reducer Phase

The reducer aggregates mutual friend counts:

1. Group all values by key (user pair)
2. If "-1" exists in values, skip this pair (already friends)
3. Otherwise, count the number of mutual friends (value count)
4. For each user, sort candidates by mutual friend count (descending)
5. Output top 10 recommendations per user

Reducer output format: `<UserID><TAB><Rec1:Count1,Rec2:Count2,...>`

3.3 Distributed Implementation

Infrastructure:

- 3 Mapper instances (t2.micro)
- 6 Reducer instances (t2.micro)
- All instances deployed in separate EC2 instances as required

Execution workflow:

1. Split input data (4.8M users) into 3 chunks
2. Upload chunks to mapper instances via SCP
3. Execute mappers in parallel, generating key-value pairs
4. Download mapper outputs
5. Partition mapper outputs by hash function across 6 reducers
6. Upload partitions to reducer instances
7. Execute reducers in parallel
8. Download and merge reducer outputs
9. Generate final recommendations

3.4 Friend Recommendation Results

Table 2 shows recommendations for the required users:

Table 2: Friend Recommendations for Report Users

User ID	Recommended Friends
924	439, 2409, 6995, 11860, 15416, 43748, 45881
8941	8943, 8944, 8940
8942	8939, 8940, 8943, 8944
9019	9022, 317, 9023
9020	9021, 9016, 9017, 9022, 317, 9023
9021	9020, 9016, 9017, 9022, 317, 9023
9022	9019, 9020, 9021, 317, 9016, 9017, 9023
9990	13134, 13478, 13877, 34299, 34485, 34642, 37941
9992	9987, 9989, 35667, 9991
9993	9991, 13134, 13478, 13877, 34299, 34485, 34642, 37941

Observations:

Users in the 9000-9100 range show overlap in recommendations (e.g., users 9019-9022 share common recommendations). User 9990 and 9993 share 7 common recommendations. Some users have fewer than 10 recommendations due to limited second-degree connections.

4 Infrastructure as Code and Automation

All infrastructure provisioning and experiment execution is fully automated:

4.1 Automation Scripts

Part 1 Execution:

```
./scripts/bootstrap_env.sh        # Setup AWS credentials
set -a; source .env; set +a       # Load environment
./run_part1.sh                     # Execute full pipeline
```

Part 2 Execution:

```
./run_part2.sh                     # Execute MapReduce pipeline
```

Cleanup:

```
./stop.sh                          # Terminate all instances
```

4.2 Automation Details

EC2 instances are created programmatically using boto3. Mappers and reducers run in parallel. SSH connections have retry logic for transient network failures. Instances are automatically terminated after completion to minimize costs. All results, plots, and metadata are saved automatically.

5 Lessons Learned

For small datasets under 1 GB, MapReduce frameworks add significant overhead without much benefit. Spark's 3.7× speedup over Hadoop shows the value of in-memory processing. Hash-based partitioning helps distribute load evenly across reducers.

SSH/SCP operations added overhead for data transfer between instances. Transient network failures required retry mechanisms. Some reducers processed more data than others due to uneven user distribution.

The friend recommendation algorithm uses a pattern where keys represent user pairs, and values represent either existing friendships (-1) or mutual friends. Reducers must handle multiple value types for the same key.

6 Conclusions

This assignment provided experience with MapReduce on AWS cloud infrastructure. Simple operations work better with lightweight tools like bash, while Spark and Hadoop are useful for larger datasets. Spark is 3.7× faster than Hadoop due to in-memory processing, though both have initialization overhead.

The friend recommendation algorithm fits well into the MapReduce paradigm with map phase emitting potential connections and reduce phase aggregating mutual friends. Automated provisioning helps ensure reproducibility.

7 Instructions to Run the Code

7.1 Prerequisites

- AWS account with appropriate permissions (EC2, VPC)
- AWS credentials configured
- Python 3.8+
- SSH key pair for EC2 access
- soc-LiveJournal1Adj.txt dataset (place in `data/` directory)

7.2 Setup

```
# Clone repository and navigate to lab2 directory
cd lab2/

# Configure AWS credentials
./scripts/bootstrap_env.sh

# Load environment variables
set -a; source .env; set +a
```

7.3 Part 1: WordCount Benchmarking

```
# Run complete Part 1 pipeline (~40-50 minutes)
./run_part1.sh

# View results
cat artifacts/summary_statistics.json
open artifacts/plot_method_comparison.png
```

7.4 Part 2: Friend Recommendation

```
# Ensure dataset is available
ls data/soc-LiveJournal1Adj.txt

# Run complete Part 2 pipeline (~15-20 minutes)
./run_part2.sh

# View recommendations
cat artifacts/report_recommendations.txt
```

7.5 Cleanup

```
# Terminate all EC2 instances
./stop.sh
```

7.6 Manual Execution (if automation fails)

```
# Part 1
python scripts/provision_wordcount.py
sleep 30
python scripts/setup_hadoop_spark.py
python scripts/run_wordcount_benchmarks.py
python plots/generate_plots.py

# Part 2
python scripts/provision_mapreduce.py
sleep 30
python scripts/deploy_mapreduce.py
python scripts/run_friend_recommendation.py
```

Appendix: Code Structure

```
lab2/
 app/
    mapper.py           # Friend recommendation mapper
    reducer.py          # Friend recommendation reducer
 wordcount/
    hadoop_wordcount.sh
    spark_wordcount.py
    linux_wordcount.sh
 scripts/
    bootstrap_env.sh
    provision_wordcount.py
    provision_mapreduce.py
    setup_hadoop_spark.py
    deploy_mapreduce.py
    run_wordcount_benchmarks.py
    run_friend_recommendation.py
    teardown.py
 plots/
```

```
   generate_plots.py
artifacts/              # Results directory
   benchmark_results.json
   summary_statistics.json
   friend_recommendations.txt
   report_recommendations.txt
   plot_*.png
run_part1.sh           # Part 1 automation
run_part2.sh           # Part 2 automation
stop.sh                # Cleanup script
README.md
```