MapReduce, Hadoop, and Spark Technologies for Cloud Computing in the Context of Federated Learning

Sina Hakimzadeh

April, 2025

1 Part 1: Foundational Knowledge & Environment Setup

In this section I define some technology that is used during the assignment; such as, MapReduce, Hadoop and spark.

1.1 MapReduce

MapReduce is a library that gives the ability to write a simple program by defining a Map and Reduce function to be run on a distributed system without knowing anything about the low-level details. It will schedule and manage the process and will also handle fault tolerance. This helps programmers to write efficient codes very fast. Here is the steps behind the programming model:

1. **Input splitting:** Before MapReduce, input data split into chunks that each process independently.
2. **Map phase:** Each Map function takes an input key-value pair and produces intermediate key-value pairs.
3. **Shuffle phase:** The system shuffles all the intermediate (key, value) pairs so that all values with the same key go to the same reducer. Then it sorts the keys to helps reducer function.
4. **Reduce phase:** Now, each reducer processes one key and all of its associated values.
5. **Output phase:** The final reduced data is written to a distributed storage. Also each reducer writes its own output file.

1.2 Hadoop

Apache Hadoop is an open-source framework designed to store and process large-scale data across a distributed cluster of computers. It is build to handle very large data, fault tolerance and job parallelism.

1.2.1 HDFS

Hadoop distributed file system is the storage layer of Hadoop. It split large data into blocks and distribute them across multiple machines; additionally, each block is replicated to ensure fault tolerance and high availability. It consists of *NameNode* which is the master that keeps tack of metadata and *DataNodes* which is the the workers that actually store the file blocks.

1.2.2 YARN

YARN(Yet Another Resource Negotiator) is the resource management and job scheduling layer of Hadoop. It manages CPU, RAM, and other resources across the cluster and Assigns containers (units of resource) to run applications. YARN also consists of *ResourceManager* which acts as the master and global manager, *NoeManager* which is workers on each node that manage local resources and run tasks and finally *ApplicationMaster* that each app has its own one to negotiates resources from ResourceManager and manage job's life cycle. Also MapReduce runs as a YARN application and request resources from YARN.

```
                              Client
                                |
                          NameNode (HDFS)
                                |
                  ---------------------------------
                  |             |                 |
              DataNode      DataNode          DataNode
                  |             |                 |
              ------------------------------------------
                                YARN
              ------------------------------------------
              ResourceManager   <--->   ApplicationMasters
                    |
               NodeManagers (1 per DataNode)
                    |
                Map & Reduce Tasks running in containers
```

1.3 Spark

Apache Spark is a fast, general-purpose distributed computing framework designed for big data processing. It was developed because of some limitations of Hadoop MapReduce, especially around speed and flexibility. It is much faster than Hadoop(up to x100 faster in memory and x10 on disk). Spark supports more than just batch processing; such as, streaming, machine learning, graph processing, etc.

1.3.1 RDDs(Resilient Distributed Datasets)

RDDs are fundamental data structure in Spark which is immutable. This data structure is distributed collections of objects that can be operated in parallel. The key concept of RDDs is *Lazy evaluation*. It means that this data structure does not execute the operation as long as it can. The user defines some transformation such as *map*, and spark will execute the operation when it is triggered by an action function such as *reduce*. This approach helps spark handle fault tolerance very efficiently.

1.3.2 DataFrame

DataFrames are Higher-level abstraction built on top of RDDs. DataFrame is a distributed collection of data organized into named columns, like a table in a relational database or a Pandas DataFrame in Python. Unlike RDDs, DataFrames has a schema like column and types, they also have higher performance because of using Catalyst optimizer. It is easier to use duo to SQL-like operation. Also Spark knows the column names and types, so it can optimize queries and catch type-related errors early.

Feature	**Spark**	**Hadoop MapReduce**
Speed	Fast (in-memory)	Slower (disk-based)
Ease of use	Easy (APIs + SQL)	Verbose Java code
Flexibility	Unified for many workloads	Batch-only
Data structure	RDDs / DataFrames	Key-value pairs

Table 1: Comparison between Spark and Hadoop MapReduce

1.3.3 Spark modes

Apache Spark can operate in several different modes depending on the environment in which it is running. The local mode is used for testing and development, where Spark runs on a single machine and processes data locally. Standalone mode allows Spark to run on multiple nodes in a cluster, managed by its own cluster manager, without relying on Hadoop. In YARN mode, Spark leverages Hadoop's YARN (Yet Another Resource Negotiator) to manage resources and run Spark applications across a cluster, while Mesos mode allows Spark to use Apache Mesos as the cluster manager for resource allocation. Finally, in Kubernetes mode, Spark can run on a Kubernetes cluster, making use of Kubernetes' powerful scheduling and resource management features. Each mode provides flexibility depending on the scale, resource management needs, and infrastructure available for running Spark jobs.

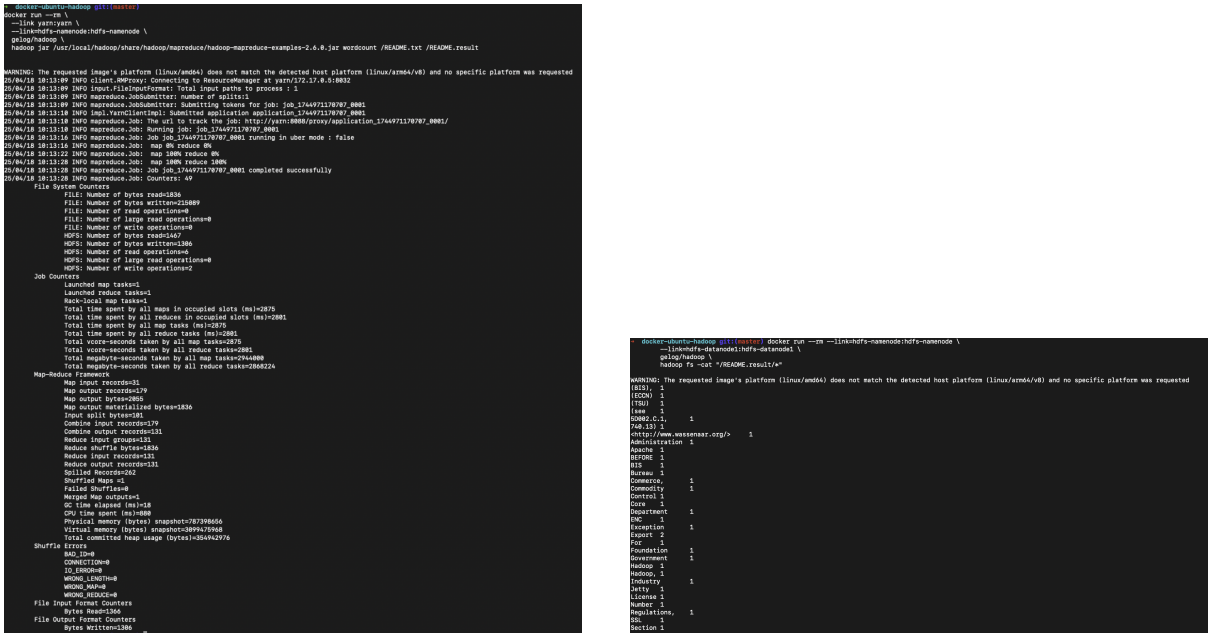
1.3.4 An advantage

Apache Spark can do its computation in memory which is a perfect choice for implementing iterative algorithms such as machine learning and AI. Because of its speed in processing large amount of data, it is a good choice.

1.4 Relationship between Hadoop and MapReduce

MapReduce is a processing model that relies on Hadoop to provide the necessary infrastructure for distributed computing. Hadoop acts as the execution environment in which MapReduce jobs run. Specifically, Hadoop handles the storage of large datasets through HDFS and manages computation resources through YARN. When a MapReduce job is launched, Hadoop splits the input data in HDFS into chunks and distributes them across the cluster. Then, it assigns map and reduce tasks to nodes based on availability and proximity to the data. In this architecture, MapReduce defines the computation logic (how data is mapped and reduced), while Hadoop takes care of job scheduling, data distribution, fault tolerance, and system coordination. Without Hadoop, MapReduce would not have the means to run at scale, as it depends entirely on Hadoop's storage and resource management capabilities.

1.5 Environment Setup



(a) Run a MapReduce example on README file

(b) Part of the Result

Figure 1: MapReduce example and its corresponding result in Hadoop

Figure 2: Running a task to find pi using Monte Carlo method with 10 tasks

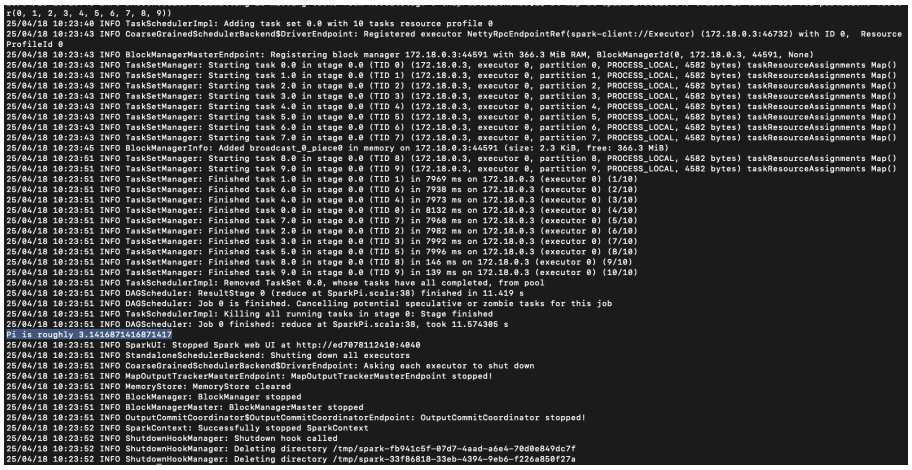


Figure 3: Result that shows the pi approximation

2 Part 2: Comparative Analysis

2.1 Processing Speed and the Reasons Behind the Differences

Hadoop processes data using the MapReduce paradigm, which involves writing intermediate results on disk between each stage. This leads to high disk I/O overhead, especially in multi-stage or iterative jobs. Spark, on the other hand, performs in-memory computations using RDDs. This reduces the need for a lot of disk writes and significantly speeds up processing, especially for iterative algorithms like those used in machine learning or graph processing.

2.2 Data Handling and Data Structures

Hadoop uses the HDFS to store and process data in a distributed environment. It works well for large-scale batch processing jobs but does not support in-memory data processing. Spark introduces RDDs and DataFrames to manage data. RDDs are fault-tolerant and distributed data collections that enable in-memory computing. DataFrames offer optimized execution plans and easier manipulation of structured data. This allows Spark to handle both structured and unstructured data more efficiently and flexibly than Hadoop.

2.3 Typical Use Cases

Hadoop is ideal for large-scale **batch** processing tasks where speed is not a critical factor, such as log processing, data archiving, etc. Its robust fault tolerance and simple design make it suitable for jobs where tasks run independently and do not need iterative operations. Spark better in more versatile workloads, including real-time data processing, machine learning, streaming analytics, and interactive queries. Its support for multiple APIs and libraries makes it a better fit for complex data pipelines and iterative tasks.

3 Part 3: Practical Application in Federated Learning

3.1 Principles of Federated Learning

Federated learning is a machine learning technique to train a model distributed across multiple servers. Instead of training a model locally on a server or device, the data are distributed over multiple machines and they train their local model using their specific data and finally update the model at the end. First, central server sends global model to client devices, then each client trains the model using its own local data and sends back the updated model to the central server. The server combines updates using an algorithm and update the global model. This steps continues until convergence.

Privacy-Preserving Advantages

Because of reaming raw data on the device, the data remain more secure and reduce the risk of exposure. Also, the data are not centralized, so it reduces the attack surface.

Assume that you want to train a model using data on your device, but it is important to keep your data private. So, this is how you can train the model(Refrence):

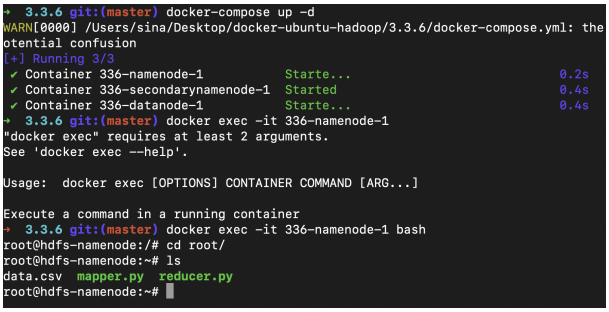
1. download the model from your source
2. train the model using your data on you own device without sharing anything.
3. only send the updated model to the cloud while it is encrypted to protect it.
4. the cloud server aggregate the data by averaging it with other users updated models.

These steps help to use sensitive data in case of privacy without storing them in a centralized storage for training models.

3.2 Hadoop: Distributed Data Processing for Initial Aggregation

I used a pseudo Hadoop docker container to implement Hadoop and MapReduce. In Part 1 we discussed how we set up the environment.

1. We use `docker-compose` to start our docker image and Hadoop environment. Then copy mapper, reducer and data file into docker by `docker cp file/path docker/path` that i copied it into */root* directory.



2. Then i copied the input data to the Hadoop distributed file system(HDFS) by using these commands:

```
# Make an input directory
hdfs dfs -mkdir -p /user/hadoop/input

# Copy data.csv into HDFS
hdfs dfs -put /path/to/data.csv /user/hadoop/input/
```

3. Everything is ready to run the MapReduce task by using this command:

```
    hadoop jar
        ↪ /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-*.jar \
    -input /user/hadoop/input \
    -output /user/hadoop/output \
    -mapper mapper.py \
    -reducer reducer.py \
    -file /root/mapper.py \
    -file /root/reducer.py
```

This command run a map reduce task that is specified the files for Hadoop to understand. The input and output directories, mapper and reducer files and where they are located is specified in the command for Hadoop. The result will be stored in `output.txt` file.

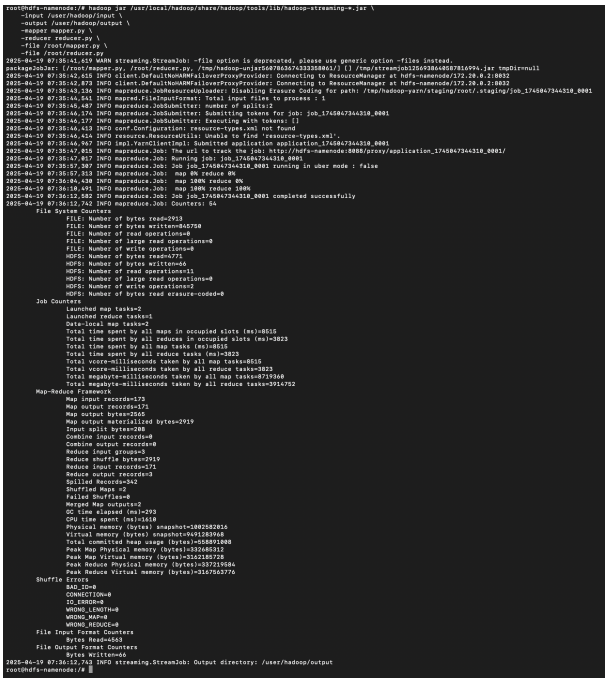


Figure 4: MapReduce task on Hadoop

To see the result, `cat` the output file from HDFS.

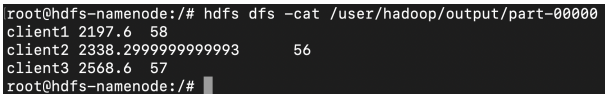


Figure 5: MapReduce output result

The result is not as expected, Why? The client2 result expected to be 57 record while it is not as expected. It is due to a bug in my mapper code which I do not consider that this code will be run on a distributed system. So i ignored the first line of data due to the column names with `header = next(reader)`. When the code is run on different partitions of data, just the first one contains column names. This leads to getting ignored some data in other partition by this line of code. To fix the problem I deleted the line and use a condition to check the line format. The mapper code provided below:

Listing 1: Mapper code

```python
#!/usr/bin/env python3
import sys
import csv

reader = csv.reader(sys.stdin)

#itertate over the rows in the CSV file
for row in reader:
    client_id = row[0] # extract the client_id
    # ignore the header line
    if client_id == "client_id":
        continue
    feature_value = float(row[1]) # extract the feature_value
    print(f"{client_id}\t{feature_value},1") # print the client_id and
        ↪ feature_value as key-value pair in hdfs
```

Listing 2: reducer code

```python
#!/usr/bin/env python3
import sys

current_id = None
sum_value = 0.0
```

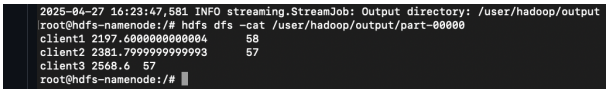
```
count = 0

# iterate over the lines from standard input whcih is the output of the mapper
# The output of the mapper is written on hdfs and
# get sorted by client_id in shuffle phase before it is passed to the reducer
for line in sys.stdin:
    # extract the client_id and feature_value from the line
    client_id, val_count = line.strip().split('\t')
    val, cnt = val_count.split(',')
    val = float(val)
    cnt = int(cnt)

    # if the client_id is the same as the current_id,
    # add the feature_value and count to the sum and count
    # otherwise, print the current_id and sum_value, and reset the sum and
        ↪ count
    if client_id == current_id:
        sum_value += val
        count += cnt
    else:
        if current_id:
            print(f"{current_id}\t{sum_value}\t{count}") # write the result of
                ↪ the previous client_id
        current_id = client_id
        sum_value = val
        count = cnt

if current_id:
    print(f"{current_id}\t{sum_value}\t{count}")
```

The final and corrected result is:



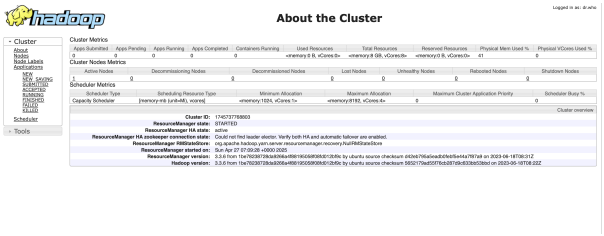


Figure 6: Hadoop UI

Hadoop provides a set of web-based user interfaces that allow users to monitor and manage their Hadoop jobs and cluster health. The Hadoop Web UI, often referred to as the ResourceManager UI and the JobHistory Server UI, provides detailed insights into the execution of MapReduce jobs, resource usage, and overall system performance. The Hadoop Web UI provides a transparent and organized way to monitor the execution of MapReduce jobs and the status of the entire cluster. It is a vital tool for administrators and users to optimize performance, debug failures, and ensure efficient resource utilization.

3.3 Spark: Global Model Aggregation and Update

In this section, I explain how I utilize Spark as a federated learning-like application to simulate its behavior. In Spark, we create some RDDs with some transformations that are ready to be run when the trigger (action) comes. So when it comes, all the operation execution starts.

1. Start the docker using docker compose command and copy the output data from Hadoop and PySpark code to docker container as same as the previous task.

```
➜  docker-spark git:(master) docker-compose up -d
WARN[0000] /Users/sina/Desktop/docker-spark/docker-compose.yml: the
sion
[+] Running 2/2
 ✔ Container spark-master    Started
 ✔ Container spark-worker-1  Started
➜  docker-spark git:(master) docker exec -it spark-master bash
bash-5.0# ls app
federated_avg.py  hadoop_output.txt
bash-5.0#
```

2. Then to be able to run a PySpark program i needed to install spark-submit on the docker because the docker does not contain it(Its path is not specified in the environment variables and may exist in other files of the container. I install the latest version manually). To install it, i used these command that i am given by ChatGPT to download and install the files:

```
wget https://archive.apache.org/dist/spark/\
spark-3.1.2/spark-3.1.2-bin-hadoop3.2.tgz

tar -xvzf spark-3.1.2-bin-hadoop3.2.tgz

mv spark-3.1.2-bin-hadoop3.2 /opt/spark

export SPARK_HOME=/opt/spark
export PATH=$SPARK_HOME/bin:$PATH
export HADOOP_CONF_DIR=/etc/hadoop/conf
```

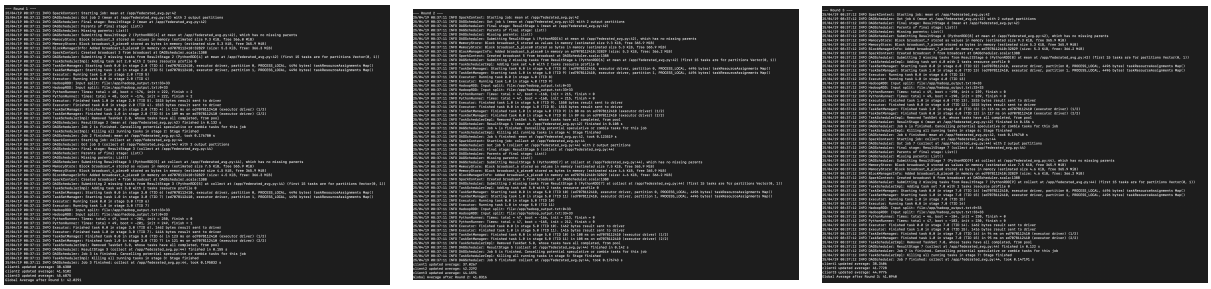
3. After the installation, i could run the program by the command below when you are in the python file directory:

```
spark-submit federated_avg.py
```

Note

The provided average is calculated with the wrong MapReduce results, and I could not change the spark results due to lake of time. I hope that you accept it.

Figure 7: Calculating the global average



(a) (b) (c)

Figure 8: All 3 rounds of simulating federated averaging

```
#Output file
Initial Global Average: 41.5468
--- Round 1 ---
client1 updated average: 38.4661
client2 updated average: 42.0802
client3 updated average: 44.9797
Global Average after Round 1: 42.1111

--- Round 2 ---
client1 updated average: 38.5865
client2 updated average: 42.6685
client3 updated average: 44.3791
Global Average after Round 2: 41.4664

--- Round 3 ---
client1 updated average: 37.2381
client2 updated average: 41.3851
client3 updated average: 44.9858
Global Average after Round 3: 41.5806
```

Listing 3: federated_avg.py code

```
from pyspark import SparkContext
import random

# Init Spark context: the entry point for Spark applications, allows us to connect to a Spark cluster and create RDDs.
sc = SparkContext(appName="FederatedAveraging")

# input file path of HDFS outpuy
file_path = "/app/hadoop_output.txt"
output_log_path = "/app/output.txt"

lines = sc.textFile(file_path)

# This function convert each line into the mentioned format: (client_id, (sum, count))
def parse_line(line):
    parts = line.strip().split("\t")
    client_id = parts[0]
    feature_sum = float(parts[1])
    count = int(parts[2])
    return (client_id, (feature_sum, count))

client_data = lines.map(parse_line)

# Read features and counts as RDDs and setting a map: Transformation
# Calculate the global average bu summing all the feature sums and counts: Action
total_sum = client_data.map(lambda x: x[1][0]).sum()
total_count = client_data.map(lambda x: x[1][1]).sum()
global_average = total_sum / total_count
```

```
28
29  log_lines = []
30  log_lines.append(f"\nInitial Global Average: {global_average:.4f}\n")
31  print(f"\nInitial Global Average: {global_average:.4f}\n")
32
33  for round_num in range(1, 4):
34      log_lines.append(f"--- Round {round_num} ---\n")
35      print(f"--- Round {round_num} ---")
36
37      # Calculate local averages
38      local_averages = client_data.map(lambda x: (x[0], x[1][0] / x[1][1]))
39
40      # Adding some noise to the local averages.
41      updated_averages = local_averages.mapValues(lambda avg: avg +
            ↪ random.uniform(-1, 1))
42
43      # Recalculate global average
44      # map : Transformation & mean: Action
45      # The map function is used to extract the updated averages, and the
            ↪ mean function calculates the average of these values.
46      global_average = updated_averages.map(lambda x: x[1]).mean()
47
48      for client_id, updated_avg in updated_averages.collect():
49          line = f"{client_id} updated average: {updated_avg:.4f}"
50          print(line)
51          log_lines.append(line + "\n")
52
53      log_lines.append(f"Global Average after Round {round_num}:
            ↪ {global_average:.4f}\n\n")
54      print(f"Global Average after Round {round_num}:
            ↪ {global_average:.4f}\n")
55
56  sc.stop()
57
58  # Save log to file
59  with open(output_log_path, "w") as f:
60      f.writelines(log_lines)
```

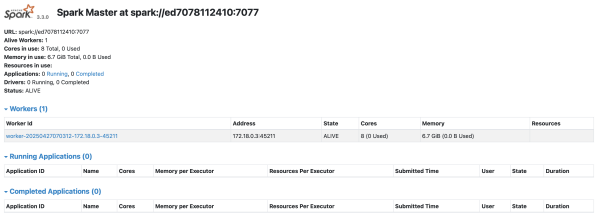


Figure 9: Spark UI

Apache Spark provides a rich set of debugging and monitoring tools through its Web UI. This interface gives a detailed view of the execution of Spark jobs, which can help in understanding performance bottlenecks, tracking progress, and debugging failures. When Spark runs, it automatically launches a Web UI on a specific port where users can monitor their applications in real-time.

Using Spark's Web UI is crucial for efficient debugging, tuning performance, and getting insights into how Spark executes your code under the hood. It turns the invisible internal processes of distributed computing into something you can see, explore, and optimize.

3.4 Analysis

In this part we discuss about the performance and benefits of using these technologies within the context of Federated Learning.

3.4.1 Performance and scalability of Hadoop

This table shows the execution time of the entire process:

Timing Type	Meaning	Interpretation in Your Case
`real`	Total wall-clock time from start to finish	**30.578 seconds** to complete the full MapReduce job, including Hadoop's job orchestration, container startup, data shuffling, etc.
`user`	CPU time spent in user space (i.e., your Python mapper and reducer logic)	**6.624 seconds** were spent executing your mapper and reducer scripts. This reflects the actual computation time.
`sys`	CPU time spent in kernel space (system calls like I/O operations)	**0.918 seconds** were spent reading from stdin and writing to stdout, typically for file access and inter-process communication.

Table 2: Analysis of Hadoop Streaming MapReduce Job Timing

As you can see in the profiling result, The map and reduce programs only takes 6.624 seconds which means about 21.6% of the entire process. It means that container overhead(simulated Hadoop), the I/O, data shuffle, and some of these operations spend the most time. Writing the map result on HDFS is also included, which is helpful and also reduces the performance. Writing on disk means that if the reduce fails, it can be re-executed and read the data from disk; on the other hand, writing on disk is an expensive job and reduces the performance. The ratio of `user/sys` to `real` indicates the bottleneck is not computation, but Hadoop job coordination and overhead.

Note

Since we are running inside a Docker container, the startup time of the Hadoop framework takes a significant portion of time.

To understand the effect of shuffle and reduce phases in this case, we disable each one to see how it changes. By running this command, the reduce phase get disabled, also shuffle phase get disabled too.

```
hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-*.jar \
-input /user/hadoop/input \
-output /user/hadoop/map_only_output \
-mapper mapper.py \
-file /root/mapper.py \
-numReduceTasks 0
```

Timing Type	Time (seconds)
Real (Total time)	23.467
User (Mapper execution)	6.520
Sys (System-level activities)	0.870

Table 3: Map Phase Timings Without Reduce

As Table 3 shows, The execution time of shuffle and reduce phases in our case is something about 7-8 seconds which means that the bottleneck is not these operations. The main bottleneck in our case is for the container and scheduling because the Hadoop is somehow get simulated and this leads to a lot of overhead. But the thing that is considerable is that the mapper execution takes 6.52 seconds, this means that the map execution over the entire map and reduce execution is about 98%. I think mapper has more execution time coming from writing on disk and also working with larger size of data and analyzing.

But what happens when the data is much larger? As we saw above, the overhead and bottleneck of operations were related to the startup and scheduling. These operations(scheduling, etc) usually take a certain amount of time. That is true if we work on a larger data set, its scheduling would take more time, but that is not comparable to the amount of time that execution takes. So mapper and reducer execution time ratio will increase by analyzing larger data set, which means better performance and utilization.

Resource utilization of Hadoop: The CPU usage of the NameNode is only high for a short period due to scheduling the MapReduce jobs on the DataNodes. During this time, the network I/O increases as it transmits tasks and data to the DataNodes. Disk reads also rise as the data is read for transmission.

The DataNodes, however, have higher CPU usage because they are responsible for executing the mapper and reducer tasks. The network I/O usage is related to receiving tasks from the NameNode. As shown in Figure 11, the map writes its results and key-value pairs to disk.

Figure 10: Namenode Performance

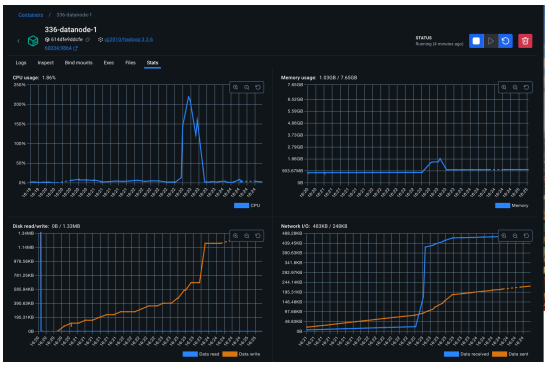


Figure 11: Datanode Performance

3.4.2 Performance and scalability of Spark

Figure 12: Spark resource utilization, As the figure shows the CPU usage, memory usage and reading data from disk

The entire Spark execution time is **10.623 seconds** which contains reading Hadoop output file and calculating the average of the model. Then use three loop iterations to simulate a federated learning. This means that instead of using multiple device to request for model, The spark send the model to them and receive the updated version, we just simulate the process using a loop. That is obvious if we implement the real operation on multiple devices, the result of profiling would be different. The spark can not distribute a python code on different devices due to some limitation, so to achieve this we must implement our application using Java or Scala. In this implementation the code just gets run on master machine.

In a larger scale number of clients the code execution time increases linear due to running it sequentially on master. But if we implement it on Java and distribute the code on some workers, some tasks such as sending and receiving the model get done simultaneously. Simultaneous execution also needs some synchronization points that leads to some inefficiency. Also the master must control the update of the model and use a different updating algorithm compare to the sequential one. But thanks to Spark these implementation can be done easily because some scaling, distributing the model and code, checking status of each worker and fault tolerance is handled by spark, without any need for understanding what is going on under the hood.

To compare Hadoop and Spark:

1. **Architecture and Purpose:** Hadoop is used for batch processing, breaking tasks into smaller ones on datanodes and reducing results. Spark, an in-memory framework, handles fault tolerance and distributes tasks while keeping intermediate data in memory to minimize disk-CPU data transfer time.

2. **Fault Tolerance:** Hadoop relies on disk writes for fault tolerance, which is slow. Spark uses RDDs to maintain data integrity during node failures without expensive disk writes.

3. **Data Processing and Performance:** Hadoop's batch processing writes intermediate results to disk, introducing latency. Spark processes data in memory, reducing disk I/O time and improving performance, especially for iterative tasks.

4. **Execution Engine:** Spark uses a Directed Acyclic Graph (DAG) execution engine, enabling fine-grained optimizations, unlike Hadoop, which lacks such an optimized execution model.

In conclusion, Spark is a good framework for iterative processes like updating the model due to its speed and fault tolerance while keeping its speed high. Hadoop is good for batch computation and model creation due to its strong fault tolerance and keeping intermediate data in disk that if one machine or cluster which are processing data goes down, we do not lose the data.

Benefits and drawbacks of using Spark and Hadoop is mentioned above and is discussed. Writing about the benefits and drawbacks of using each one would be the same. But The only additional point is, using spark is a better choice for federated learning updates tasks,why? request-based and real-time application is more suitable for Spark due to in memory processing of sparks. It can distribute tasks and then gather the updated result of each device to update the main model without need of writing a lot of data on disk. Also its RDDs and queue of operations(transformations) helps a lot to recover failure in the system and re-run the operation for the specific failure.

3.4.3 Federated Learning Challenges and Benefits

The following responses address each objective in order:

1. In our tasks we implemented a pipeline for federated learning that contains a model creation using Hadoop and a federated learning simulation using Spark. Hadoop is a good choice for analyzing data and extract the model(model creation) from the data, but as i mentioned before, its startup and writing on disk is a costly operation, but for training a model, it is not vital to be fast. In Federated learning, all the data is partitioned between a lot of nodes that are devices with their data. Spark is a good choice to act as a master node and consider other devices as many workers. Spark must handle a lot of updated model that came from workers and update the main model. The bottleneck in federated learning in this case is the overhead of communication between Spark and devices. Spark must handle this overhead while considering the fault tolerance and do not let a device create an stall in the system. It must handle the situation where a device are fails and cannot return the updated model.

2. The most important benefit of using these technologies is handling large amounts of data and their computations. Thanks to these technologies, we can process vast datasets in distributed environments and create a large-scale parallel system that responds quickly and can be scaled based on demand. As I mentioned above, federated learning involves having a lot of data distributed across different nodes, and we must handle it efficiently. MapReduce can analyze large data to extract training data quickly by writing only a few lines of code, while also handling fault tolerance and other related concerns. Spark also handles the federated learning operations, including creating and updating the model efficiently.

3. For the initial data aggregation, Hadoop is a good choice because it can process large amounts of raw data using MapReduce. This helps us prepare and organize the training data with a simple and fault-tolerant system. The job is split across multiple nodes, so it's easy to scale, and it handles failures automatically. However, Hadoop writes intermediate data to disk, which makes it slower, especially when we don't need multiple passes over the data. For global model aggregation, Spark is more efficient because it uses in-memory processing. Since federated learning involves combining updates from different clients in several rounds, using Spark speeds up the process by avoiding repeated disk I/O. It also makes the coding easier with high-level APIs. But Spark needs more memory to work efficiently, and setting it up can be more complex compared to Hadoop.

4 Conclusion

In conclusion, this analysis has provided an in-depth comparison between Hadoop and Spark in the context of distributed data processing, particularly with respect to federated learning. While Hadoop's MapReduce paradigm offers a robust framework for large-scale batch processing with fault tolerance, its

reliance on disk-based operations results in higher overhead and slower processing times compared to Spark. Spark, with its in-memory processing and advanced data structures like RDDs and DataFrames, offers significant performance improvements, especially for iterative tasks and machine learning applications. Moreover, the comparison of both frameworks in a Federated Learning scenario highlights the efficiency of Spark for real-time and iterative processes, while Hadoop remains more suitable for simpler, batch-oriented tasks. This study demonstrates the practical implications of choosing the right tool based on the specific needs of a distributed learning environment, where Spark's flexibility and speed make it a more suitable candidate for dynamic and privacy-preserving applications like federated learning.