

Université d'El Oued

Faculté des Sciences Exactes
Département d'Informatique

Rapport de TP 9

Traitement Spark Batch et Streaming avec Docker

Étudiant : azizi lazhar

Spécialisation : Master II : IA & Science des Données

Résumé

Ce rapport documente la mise en place d'un cluster Hadoop/Spark à nœud unique en utilisant Docker. Nous explorons trois paradigmes principaux de traitement des données : le traitement interactif avec Spark Shell (Scala), le traitement par lots à l'aide d'une application Java compilée, et le traitement en temps réel avec Spark Structured Streaming.

1 Objectif du laboratoire

L'objectif principal de ce laboratoire est de déployer un environnement **Big Data** conteneurisé et d'exécuter des tâches de traitement de données distribuées. Le laboratoire comprend :

- Le déploiement d'un cluster **Hadoop/Spark** en utilisant **Docker Compose**.
- L'exécution d'analyses de données interactives avec **Scala (Spark Shell)**.
- Le développement et la soumission d'un travail de traitement par lots en **Java (MapReduce)**.
- La mise en oeuvre d'une chaîne de traitement en temps réel utilisant **Spark Structured Streaming**.

2 Installation et configuration du cluster

2.1 Initialisation de l'environnement

Nous avons initialisé le cluster en utilisant **Docker Compose**. Après le déploiement, nous avons configuré les clés **SSH** et les autorisations des utilisateurs pour assurer une communication fluide entre le **NameNode** et les **DataNodes**..

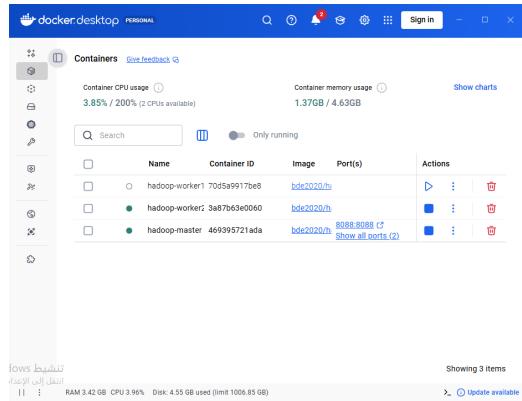


FIGURE 1 – Docker Compose.

```
Administrator: ~ docker exec -it hadoop-master bash
(c) Microsoft Corporation. 2016. Tous droits réservés.

C:\Windows\system32>docker start hadoop-master hadoop-worker1 hadoop-worker2
hadoop-master
hadoop-worker1
hadoop-worker2

C:\Windows\system32>docker exec -it hadoop-master bash
root@hadoop-master:/# docker exec -it hadoop-master bash
bash: docker: command not found
root@hadoop-master:/# ls
KEYS boot docker etc hadoop-data lib media opt root run.sh srv tmp var
bin dev entrypoint.sh hadoop home lib64 mnt proc run sbin sys usr
root@hadoop-master:/# /hadoop/sbin/start-dfs.sh
bash: /hadoop/sbin/start-dfs.sh: No such file or directory
root@hadoop-master:/# /opt/hadoop-3.2.1/sbin/start-dfs.sh
Starting namenodes on [hadoop-master]
hadoop-master: /opt/hadoop-3.2.1/bin/../libexec/hadoop-functions.sh: line 982: ssh: command not found
Starting datanodes
localhost: /opt/hadoop-3.2.1/bin/../libexec/hadoop-functions.sh: line 982: ssh: command not found
Starting secondary namenodes [hadoop-master]
hadoop-master: /opt/hadoop-3.2.1/bin/../libexec/hadoop-functions.sh: line 982: ssh: command not found
root@hadoop-master:/# jps
3110 Jps
87 NameNode
1929 NodeManager
1707 DataNode
734 ResourceManager
1791 SecondaryNameNode
root@hadoop-master:/#
```

FIGURE 2 – Démarrage du cluster : la commande jps confirme que NameNode, DataNode et ResourceManager sont actifs..

2.2 Vérification des interfaces Web

Nous avons vérifié l'état du cluster via les interfaces web **HDFS** et **YARN** pour nous assurer que les ressources étaient correctement allouées.

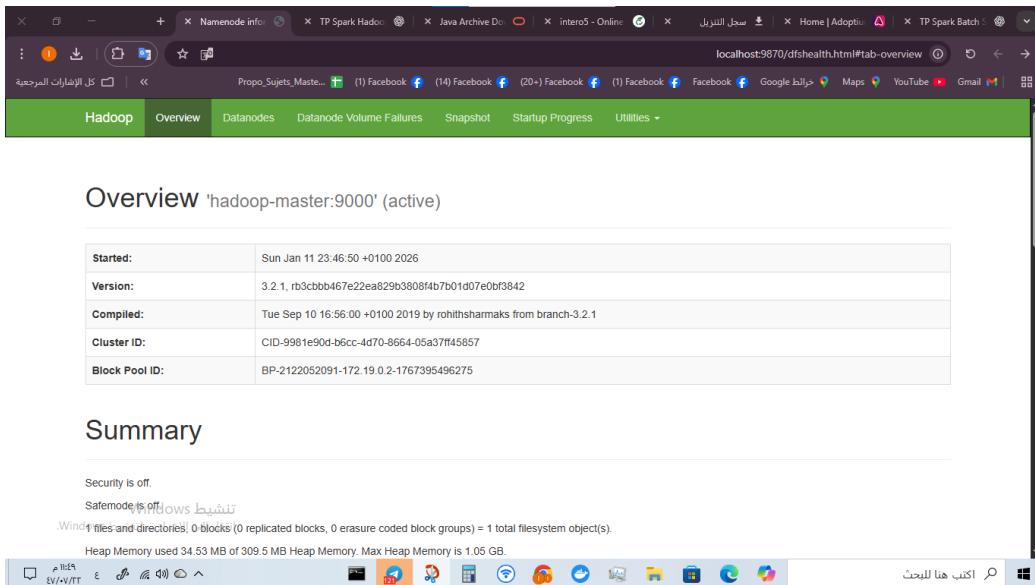


FIGURE 3 – Interface Web HDFS (Port 9870)

All Applications

ID	User	Name	Application Type	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU Vcores	Allocated Memory MB	Reserved CPU Vcores	Reserved Memory MB
Showing 0 to 0 of 0 entries															

No data available in table

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCore
0	0	0	0	0	0 B	8 GB	0 B	0

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Reb
1	0	0	0	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation
Capacity Scheduler	[memory-mb (unit=Mi), vcores]	<memory:1024, vCores:1>	<memory:8192, vCores:4>

Show 20 > entries

FIGURE 4 – Gestionnaire de ressources YARN (Port 8088)

3 Phase 1 : Traitement interactif (Spark Shell)

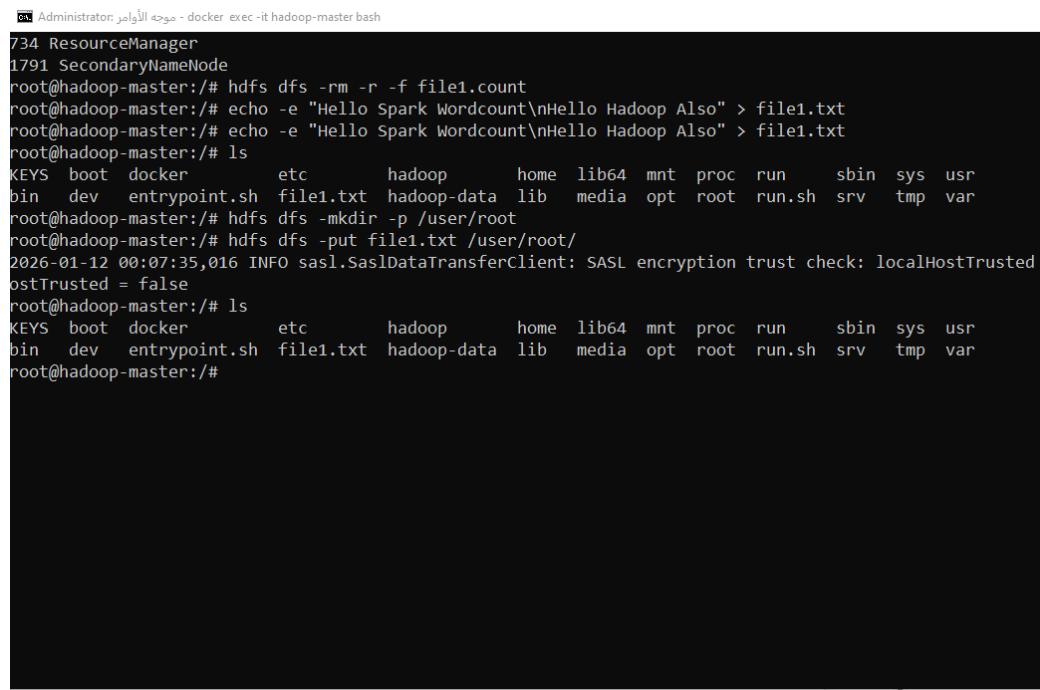
Dans cette phase, nous avons téléchargé un fichier texte sur **HDFS** et effectué un comptage de mots en utilisant **Scala** dans le **Spark Shell** inter-

actif.

3.1 Préparation des données

Nous avons créé un fichier exemple à l'intérieur du conteneur et l'avons téléchargé sur le système de fichiers distribué.

1. hdfs dfs -rm -r -f file1.count
2. echo -e "Hello Spark Wordcount\nHello Hadoop Also" > file1.txt
3. hdfs dfs -mkdir -p /user/root
4. hdfs dfs -put file1.txt /user/root/



```
Administrator: اجر چو - docker exec -it hadoop-master bash
734 ResourceManager
1791 SecondaryNameNode
root@hadoop-master:/# hdfs dfs -rm -r -f file1.count
root@hadoop-master:/# echo -e "Hello Spark Wordcount\nHello Hadoop Also" > file1.txt
root@hadoop-master:/# echo -e "Hello Spark Wordcount\nHello Hadoop Also" > file1.txt
root@hadoop-master:/# ls
KEYS boot docker etc hadoop home lib64 mnt proc run sbin sys usr
bin dev entrypoint.sh file1.txt hadoop-data lib media opt root run.sh srv tmp var
root@hadoop-master:/# hdfs dfs -mkdir -p /user/root
root@hadoop-master:/# hdfs dfs -put file1.txt /user/root/
2026-01-12 00:07:35,016 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted
ostTrusted = false
root@hadoop-master:/# ls
KEYS boot docker etc hadoop home lib64 mnt proc run sbin sys usr
bin dev entrypoint.sh file1.txt hadoop-data lib media opt root run.sh srv tmp var
root@hadoop-master:/#
```

FIGURE 5 – Téléversement des données d'entrée sur HDFS

3.2 Execution Results

Nous avons exécuté la logique **MapReduce** en utilisant l'API fonctionnelle de **Scala** :

```

1. val lines = sc.textFile("file1.txt")
2. val wc = lines.flatMap(_.split("\\s+")).map(w => (w, 1)).reduceByKey(_ + _)
3. wc.saveAsTextFile("hdfs://hadoop-master:9000/user/root/file1.count")

```

The screenshot shows the Spark Shell interface. At the top, it displays the Scala logo and the text "version 4.1.0". Below that, the Scala REPL prompt "scala>" is followed by the code for reading a file, splitting words, mapping them with a count of 1, reducing by key to sum counts, and saving the result. The output shows the execution of the code, including the creation of RDDs and their conversion to a collection of tuples. The final output is a collection of four tuples: (Hello, 1), (Spark, 2), (Hadoop, 1).

```

Using Scala version 2.13.17 (OpenJDK 64-Bit Server VM, Java 17.0.17)
Type in expressions to have them evaluated.
Type help() for information.
26/01/12 15:03:16 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
26/01/12 15:03:16 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
Spark context Web UI available at http://172.29.20.81:4041
Spark context available as 'sc' (master = local[*], app id = local-1768226597888).
Spark session available as 'spark'.

scala> val lines = sc.textFile("file:///home/lazhar/file1.txt")
val lines: org.apache.spark.rdd.RDD[String] = file:///home/lazhar/file1.txt MapPartitionsRDD[1] at textFile at <console>:1

scala> val words = lines.flatMap(_.split("\\s+"))
val words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at flatMap at <console>:1

scala> val wc = words.map(w => (w, 1)).reduceByKey(_ + _)
val wc: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:1

scala> wc.collect().foreach(println)
(Hello,1)
(Spark,2)
(Hadoop,1)

```

FIGURE 6 – Résultat de la phase 1 : exécution réussie du comptage de mots dans Spark Shell

The screenshot shows a terminal window. It starts with the command "scala> :quit" to exit the Scala REPL. Then, it runs the command "root@hadoop-master:~# hdfs dfs -cat file1.count/part-00000" which outputs the contents of the HDFS file, showing three lines: "(Hello,1)", "(Spark,2)", and "(Hadoop,1)".

```

scala> :quit
root@hadoop-master:~# hdfs dfs -cat file1.count/part-00000
(Hello,1)
(Spark,2)
(Hadoop,1)
root@hadoop-master:~#

```

FIGURE 7 – Résultat de la phase 1 : exécution réussie du comptage de mots dans Spark Shell

4 Phase 2 : Traitement par lots (Java)

Nous avons développé une application Java autonome pour traiter des données de transactions (**Date**, **Ville**, **Article**, **Prix**) stockées dans **HDFS**.

4.1 Implémentation

Le fichier **WordCountTask.java** lit des valeurs séparées par des tabulations et compte les occurrences.

```
// Phase Map
JavaPairRDD<String, Integer> counts = textFile
    .flatMap(s -> Arrays.asList(s.split("\t")).iterator())
    .mapToPair(word -> new Tuple2<>(word, 1))
    .reduceByKey((a, b) -> a + b); // Phase Reduce
```

Listing 1 – Extrait de WordCountTask.java

```
1 JavaPairRDD<String, Integer> counts = textFile
2     .flatMap(s -> Arrays.asList(s.split("\t")).iterator()
3             )
4     .mapToPair(word -> new Tuple2<>(word, 1))
4     .reduceByKey((a, b) -> a + b); // Phase Reduce
```

4.2 Compilation et Soumission

Le code a été compilé en un fichier **JAR** et soumis au cluster **Spark** en utilisant la commande **spark-submit**.

```
root@hadoop-master:~# hdfs dfs -rm -r -f out-spark
root@hadoop-master:~# hdfs dfs -mkdir -p input
root@hadoop-master:~# echo -e "2024-01-01\NY\Shoes\t100" > purchases.txt
root@hadoop-master:~# hdfs dfs -put purchases.txt input/
root@hadoop-master:~#
```

Windows تنشيط
انقل إلى الإعدادات لتنشيط dows

FIGURE 8 – Préparation du fichier d'entrée purchases.txt.

```

root@hadoop-master:~# javac -cp "/usr/local/spark/jars/*" WordCountTask.java
root@hadoop-master:~# jar cf wordcount.jar WordCountTask.class
root@hadoop-master:~# spark-submit --class WordCountTask --master local wordcount.jar \
> hdfs://hadoop-master:9000/user/root/input/purchases.txt \
> hdfs://hadoop-master:9000/user/root/out-spark 2> /dev/null

root@hadoop-master:~# hdfs dfs -cat out-spark/part-00000
(NY,1)
(100,1)
(Shoes,1)
(2024-01-01,1)
root@hadoop-master:~#

```

تنشيط Windows
انقل إلى الأعداد لتنشيط Windows

FIGURE 9 – Résultat de la phase 2 : la sortie dans HDFS montre des décomptes corrects pour les données des transactions.

5 Phase 3 : Traitement en temps réel (Streaming)

La phase finale a démontré l'utilisation de **Spark Structured Streaming** en écoutant un socket TCP sur le port 9999.

5.1 Configuration du Streaming

L'application lit les données depuis le socket et met à jour les comptes toutes les secondes.

5.2 Exemple de Streaming Scala

Voici un exemple de code pour lire un flux de données depuis un socket et afficher les résultats dans la console :

```

Dataset<Row> lines = spark.readStream()
    .format("socket")
    .option("host", "localhost")
    .option("port", 9999)

```

```

.load();

// Output to Console
StreamingQuery query = wordCounts.writeStream()
    .outputMode("complete")
    .format("console")
    .trigger(Trigger.ProcessingTime("1 second"))
    .start();

```

The screenshot shows the Spark Shell interface. At the top, it displays the Scala logo and the text "version 4.1.0". Below that, the Scala REPL prompt "scala>" is followed by the code for reading a file, splitting it into words, mapping each word to a tuple (word, 1), reducing by key to get the count for each word, and finally collecting the results. The output shows three words: "Hello" (count 2), "Spark" (count 2), and "Hadoop" (count 1). The entire session is wrapped in a try-catch block.

```

Using Scala version 2.13.17 (OpenJDK 64-Bit Server VM, Java 17.0.17)
Type in expressions to have them evaluated.
Type :help for more information.
26/01/12 15:03:10 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
26/01/12 15:03:10 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
Spark context Web UI available at http://172.29.20.81:4041
Spark context available as `sc` (master = local[*], app id = local-1768226597888).
Spark session available as `spark`.

scala> val lines = sc.textFile("file:///home/lazhar/file1.txt")
val lines: org.apache.spark.rdd.RDD[String] = file:///home/lazhar/file1.txt MapPartitionsRDD[1] at textFile at <console>:1

scala> val words = lines.flatMap(_.split("\\s+"))
val words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at flatMap at <console>:1

scala> val wc = words.map(w => (w, 1)).reduceByKey(_ + _)
val wc: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:1

scala> wc.collect().foreach(println)
(Hello,2)
(Spark,2)
(Hadoop,1)

```

FIGURE 10 – Résultat de la phase 1 : exécution réussie du comptage de mots dans Spark Shell

The screenshot shows a terminal window with a root shell on a Hadoop master node. It runs the command "hdfs dfs -cat file1.count/part-00000", which outputs the word counts: "Hello,2". The session ends with the command ":quit".

```

scala> :quit
root@hadoop-master:~# hdfs dfs -cat file1.count/part-00000
>Hello,2
root@hadoop-master:~#

```

FIGURE 11 – Résultat de la phase 1 : exécution réussie du comptage de mots dans Spark Shell

5.3 Démonstration en direct

Nous avons utilisé la commande `nc -lk 9999` pour simuler un flux de données. Au fur et à mesure que des mots étaient saisis dans le terminal, **Spark** mettait à jour la matrice de comptage en temps réel.

```

ewriter[numRows=20, truncate=true]] is committing.      root@hadoop-master:~# nc -lk 9999
Batch: 9
+-----+
| value|count|
+-----+
| RED| 1|
| green| 1|
| hello| 2|
| GREEN| 1|
| streaming| 1|
| li | 2|
| red| 1|
| data| 5|
| Green| 1|
| spark| 2|
| Blue| 1|
| BLUE| 1|
| world| 1|
| Red| 1|
| blue| 1|
| big| 1|
+-----+

```

FIGURE 12 – Exécution en temps réel : le terminal de gauche montre Spark en train de produire des lots mis à jour ; le terminal de droite montre le flux de saisie de l’utilisateur.

6 Conclusion

Ce laboratoire a démontré avec succès la polyvalence d'**Apache Spark** dans un environnement **Hadoop** conteneurisé. Nous sommes passés de commandes interactives simples à des travaux batch compilés, puis enfin au traitement de flux en temps réel, en vérifiant les résultats à chaque étape à l'aide de **HDFS** et des sorties console. L'approche conteneurisée a permis de créer un environnement **Big Data** reproductible et isolé.