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 œuvre 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**..

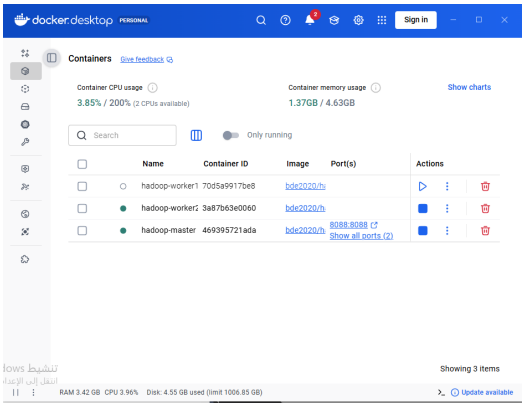

FIGURE 1 – Docker Compose.

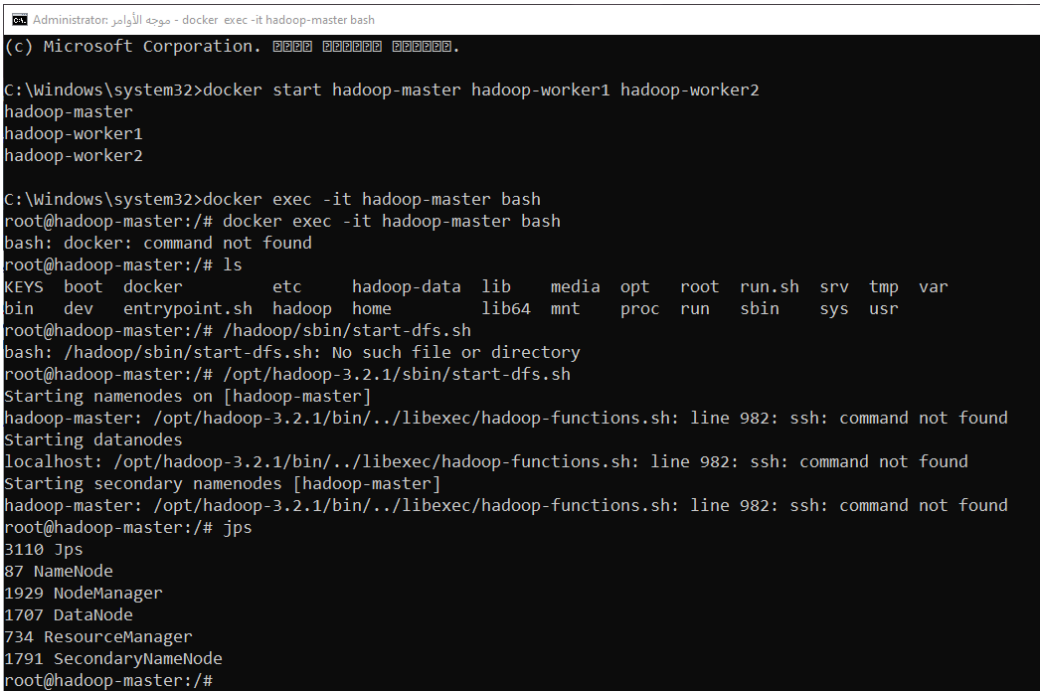

FIGURE 2 – Démarrage du cluster : la commande jps confirme que NameNode, DataNode et ResourceManager sont actifs..

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.

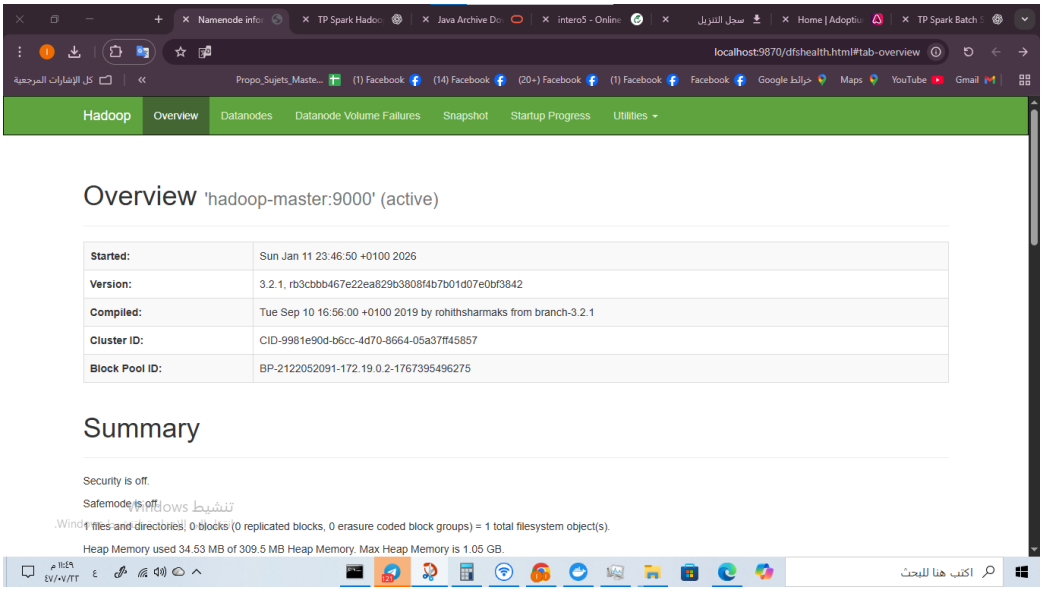

FIGURE 3 – Interface Web HDFS (Port 9870)

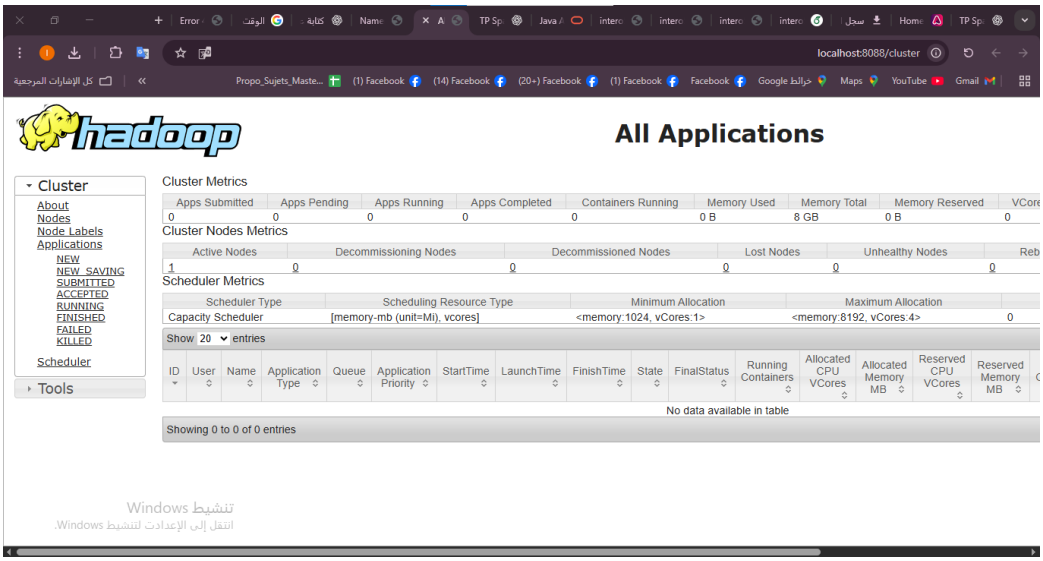

FIGURE 4 – Gestionnaire de ressources YARN (Port 8088)

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/
```

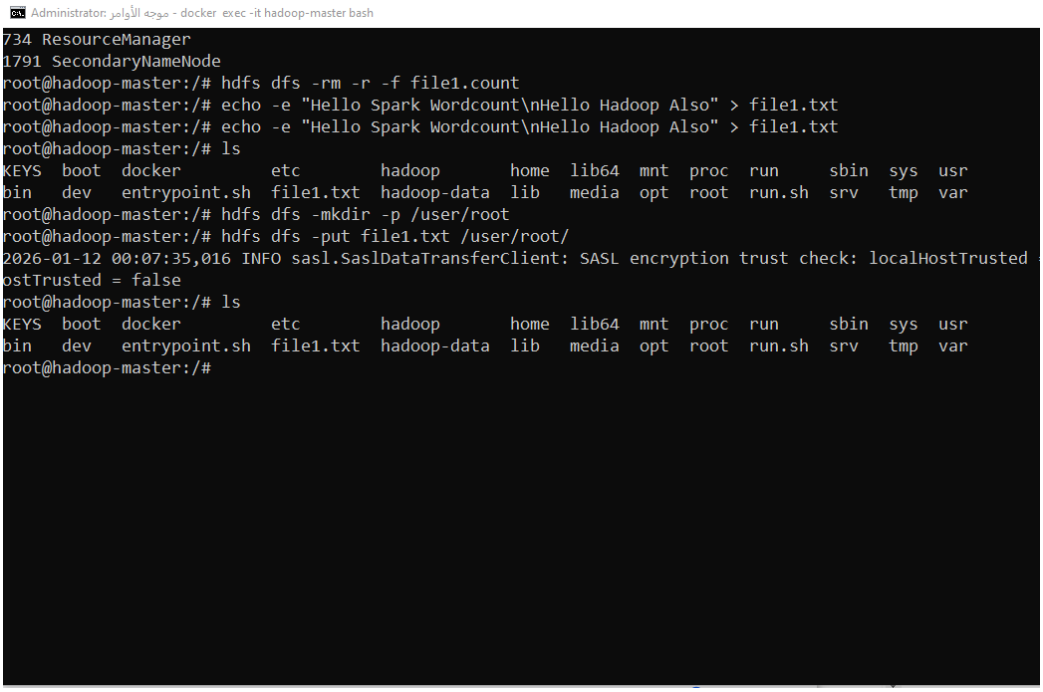

FIGURE 5 – Téléversement des données d'entrée sur HDFS

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")
```

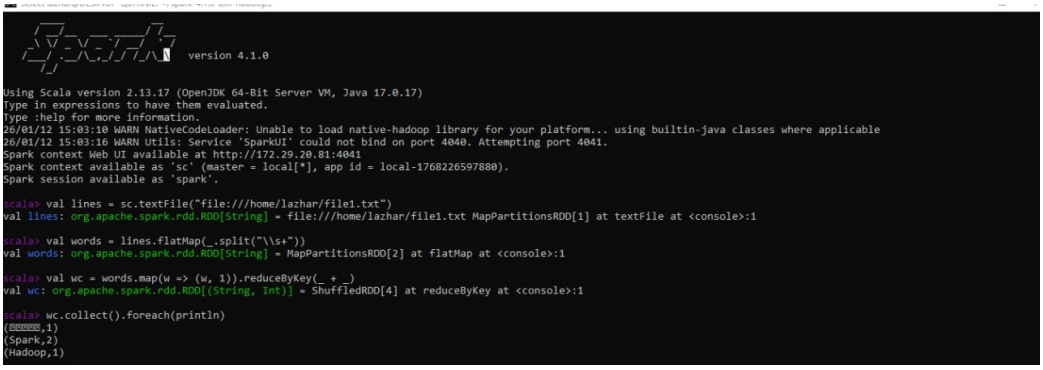

FIGURE 6 – Résultat de la phase 1 : exécution réussie du comptage de mots dans Spark Shell

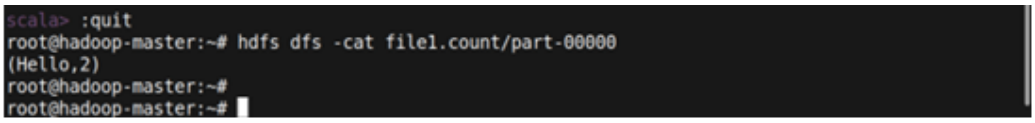

FIGURE 7 – Résultat de la phase 1 : exécution réussie du comptage de mots dans Spark Shell

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     .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`.

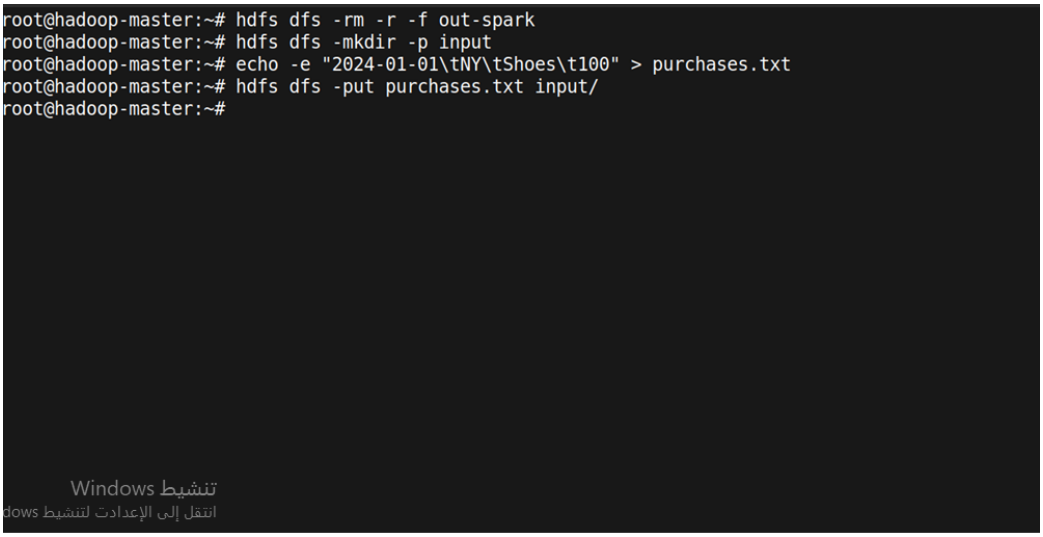

FIGURE 8 – Préparation du fichier d'entrée purchases.txt.

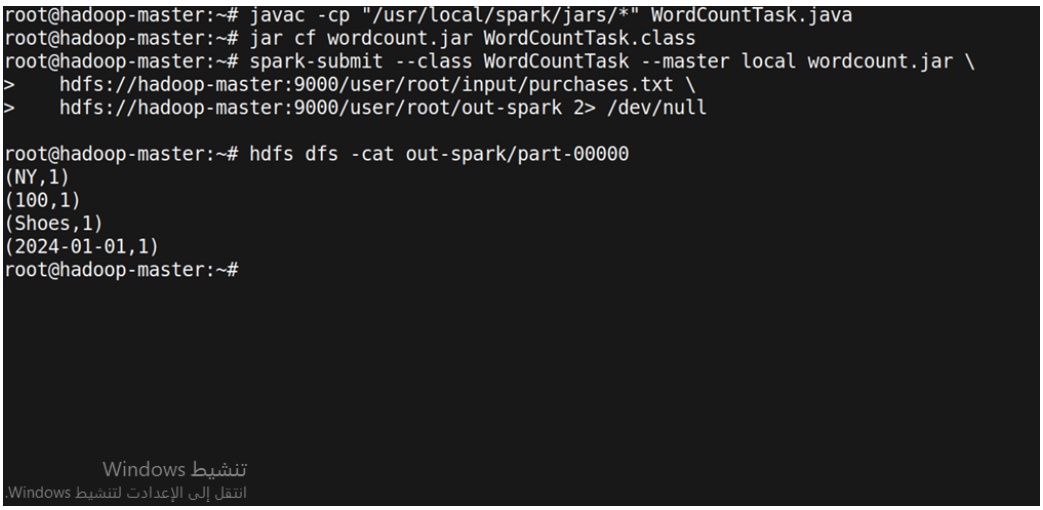

FIGURE 9 – Résultat de la phase 2 : la sortie dans HDFS montre des décomptes corrects pour les données des transactions.

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();
```

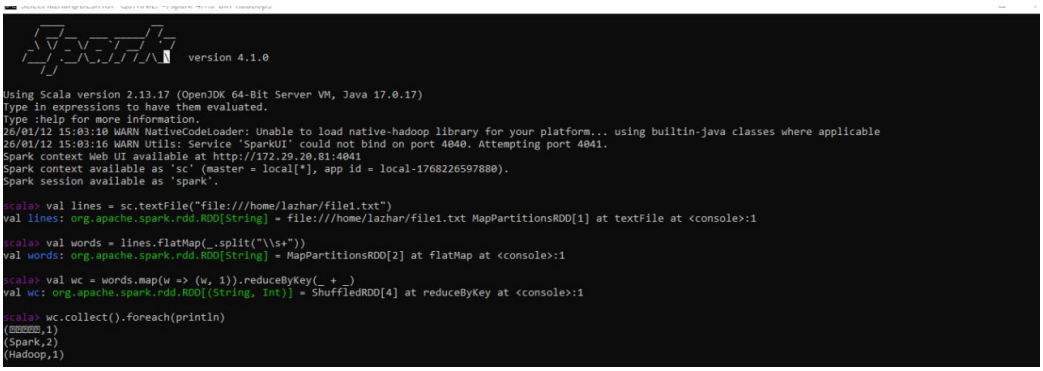

FIGURE 10 – Résultat de la phase 1 : exécution réussie du comptage de mots dans Spark Shell

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.

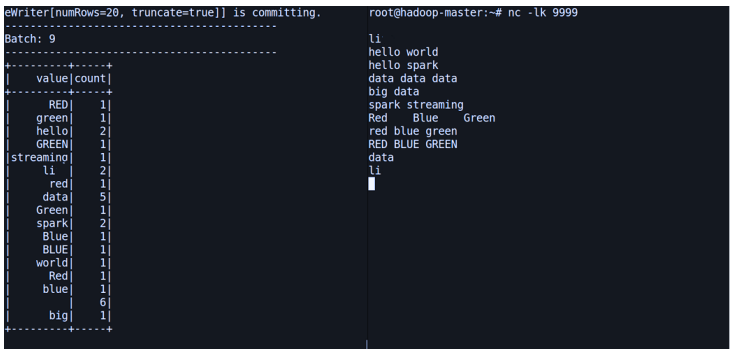

FIGURE 12 – Exécution en temps réel : le terminal de gauche montre Spark en train de produire des lots mis à jour ; le terminal de droite montre le flux de saisie de l'utilisateur.

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.