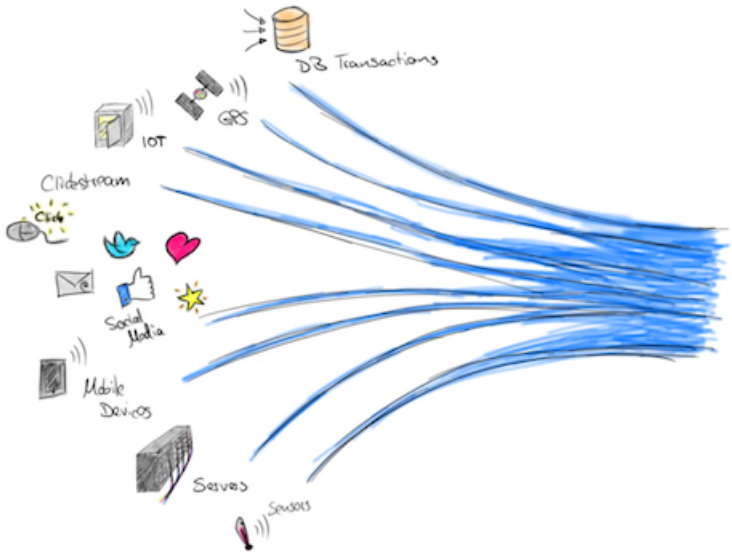
TP2 - Traitement par Lot et Streaming avec Spark



Télécharger PDF

Objectifs du TP

Utilisation de Spark pour réaliser des traitements par lot et des traitements en streaming.

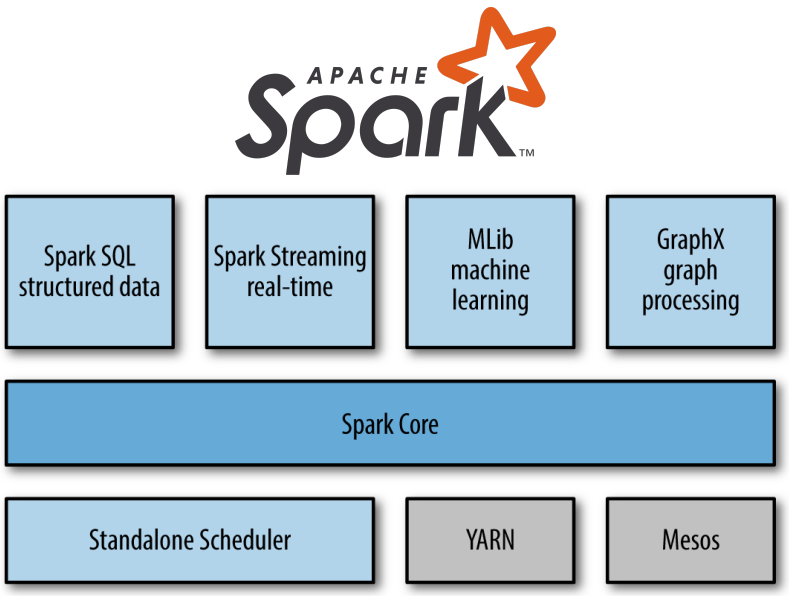
Outils et Versions

- Apache Hadoop Version: 3.3.6
- Apache Spark Version: 3.5.0
- Docker Version *latest*
- Visual Studio Code Version 1.85.1 (ou tout autre IDE de votre choix)
- Java Version 1.8.
- Unix-like ou Unix-based Systems (Divers Linux et MacOS)

Spark

Présentation

Spark est un système de traitement rapide et parallèle. Il fournit des APIs de haut niveau en Java, Scala, Python et R, et un moteur optimisé qui supporte l'exécution des graphes. Il supporte également un ensemble d'outils de haut niveau tels que Spark SQL pour le support du traitement de données structurées, MLlib pour l'apprentissage des données, GraphX pour le traitement des graphes, et Spark Streaming pour le traitment des données en streaming.



Spark et Hadoop

Spark peut s'exécuter sur plusieurs plateformes: Hadoop, Mesos, en standalone ou sur le cloud. Il peut également accéder à diverses sources de données, comme HDFS, Cassandra, HBase et S3.

Dans ce TP, nous allons exécuter Spark sur Hadoop YARN. YARN s'occupera ainsi de la gestion des ressources pour le déclenchement et l'exécution des Jobs Spark.

Installation

Nous avons procédé à l'installation de Spark sur le cluster Hadoop utilisé dans le TP1. Suivre les étapes décrites dans la partie *Installation* du TP1 pour télécharger l'image et exécuter les trois contenaires. Si cela est déjà fait, il suffit de lancer vos machines grâce aux commandes suivantes:

```
docker start hadoop-master hadoop-worker1 hadoop-worker2
```

puis d'entrer dans le contenaire master:

```
docker exec -it hadoop-master bash
```

Lancer ensuite les démons yarn et hdfs:

```
./start-hadoop.sh
```

Vous pourrez vérifier que tous les démons sont lancés en tapant: `jps`. Un résultat semblable au suivant pourra être visible:

```
880 Jps
257 NameNode
613 ResourceManager
456 SecondaryNameNode
```

La même opération sur les noeuds workers (auxquels vous accédez à partir de votre machine hôte de la même façon que le noeud maître, c'est à dire en tapant par exemple `docker exec -it hadoop-worker1 bash`) devrait donner:

```
176 NodeManager
65 DataNode
311 Jps
```

Test de Spark avec Spark-Shell

Dans le but de tester l'exécution de spark, commencer par créer un fichier *file1.txt* dans votre noeud master, contenant le texte suivant:

```
Hello Spark Wordcount!
Hello Hadoop Also :)
```

Charger ensuite ce fichier dans HDFS:

```
hdfs dfs -put file1.txt
```

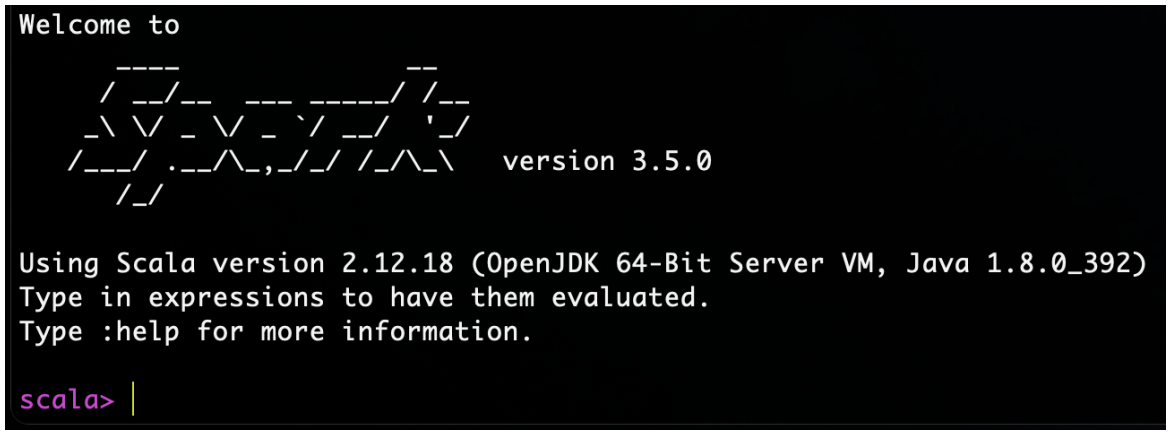
Erreur possible

Si le message suivant s'affiche: `put: `.': No such file or directory`, c'est parce que l'arborescence du répertoire principal n'est pas créée dans HDFS. Pour le faire, il suffit d'exécuter la commande suivante avant la commande de chargement : `hadoop fs mkdir -p .`

Pour vérifier que spark est bien installé, taper la commande suivante:

```
spark-shell
```

Vous devriez avoir un résultat semblable au suivant:



Vous pourrez tester spark avec un code scala simple comme suit (à exécuter ligne par ligne):

```scala
val lines = sc.textFile("file1.txt")
val words = lines.flatMap(_.split("\\s+"))
val wc = words.map(w => (w, 1)).reduceByKey(_ + _)
wc.saveAsTextFile("file1.count")
```

Ce code vient de (1) charger le fichier *file1.txt* de HDFS, (2) séparer les mots selon les caractères d'espacement, (3) appliquer un *map* sur les mots obtenus qui produit le couple (<*mot*>, 1), puis un *reduce* qui permet de faire la somme des 1 des mots identiques.

Pour afficher le résultat, sortir de spark-shell en cliquant sur *Ctrl-C*. Télécharger ensuite le répertoire *file1.count* créé dans HDFS comme suit:

```
hdfs dfs -get file1.count
```

Le contenu des deux fichiers *part-00000* et *part-00001* ressemble à ce qui suit:

```
root@hadoop-master:~/file1.count# cat part-00000
(Hello,2)
(Wordcount!,1)
root@hadoop-master:~/file1.count# cat part-00001
(Spark,1)
(:),1)
(Also,1)
(Hadoop,1)
```

L'API de Spark

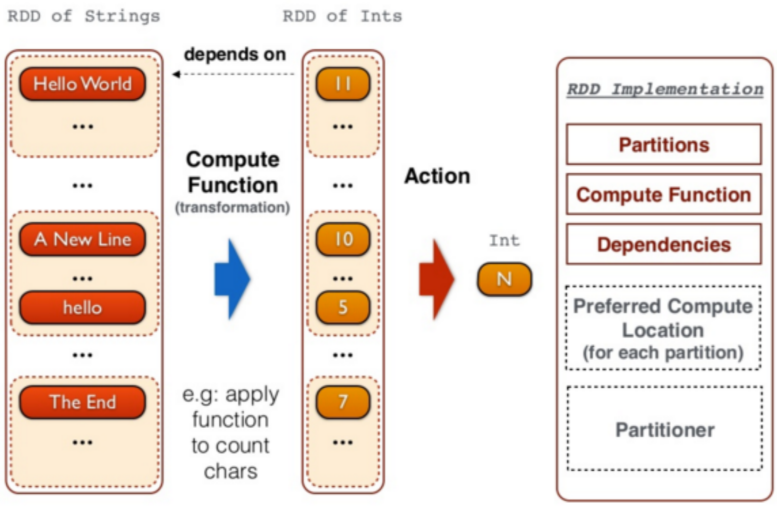
A un haut niveau d'abstraction, chaque application Spark consiste en un programme *driver* qui exécute la fonction *main* de l'utilisateur et lance plusieurs opérations parallèles sur le cluster. L'abstraction principale fournie par Spark est un RDD (*Resilient Distributed Dataset*), qui représente une collection d'éléments partitionnés à travers les noeuds du cluster, et sur lesquelles on peut opérer en parallèle. Les RDDs sont créés à partir d'un fichier dans HDFS par exemple, puis le transforment. Les utilisateurs peuvent demander à Spark de sauvegarder un RDD en mémoire, lui permettant ainsi d'être réutilisé efficacement à travers plusieurs opérations parallèles.

Les RDDs supportent deux types d'opérations:

- les *transformations*, qui permettent de créer un nouveau Dataset à partir d'un Dataset existant
- les *actions*, qui retournent une valeur au programme *driver* après avoir exécuté un calcul sur le Dataset.

Par exemple, un *map* est une transformation qui passe chaque élément du dataset via une fonction, et retourne un nouvel RDD représentant les résultats. Un *reduce* est une action qui agrège tous les éléments du RDD en utilisant une certaine fonction et retourne le résultat final au programme.

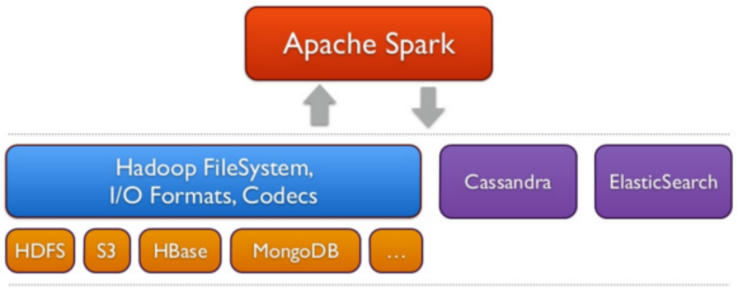
Toutes les transformations dans Spark sont *lazy*, car elles ne calculent pas le résultat immédiatement. Elles se souviennent des transformations appliquées à un dataset de base (par ex. un fichier). Les transformations ne sont calculées que quand une action nécessite qu'un résultat soit retourné au programme principal. Cela permet à Spark de s'exécuter plus efficacement.



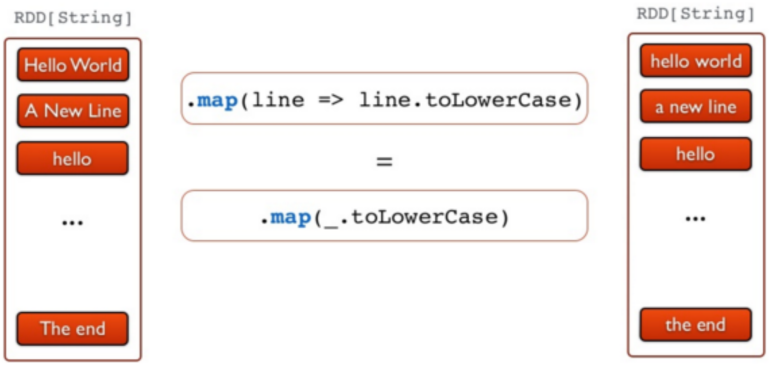
Exemple

L'exemple que nous allons présenter ici par étapes permet de relever les mots les plus fréquents dans un fichier. Pour cela, le code suivant est utilisé:

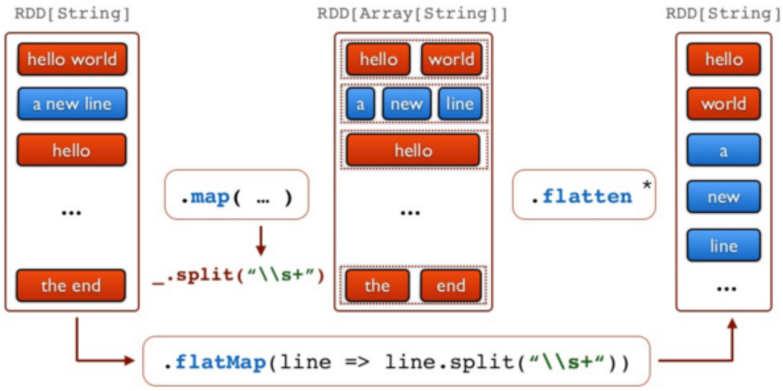
```
//Etape 1 - Créer un RDD à partir d'un fichier texte de Hadoop
val docs = sc.textFile("file1.txt")
```



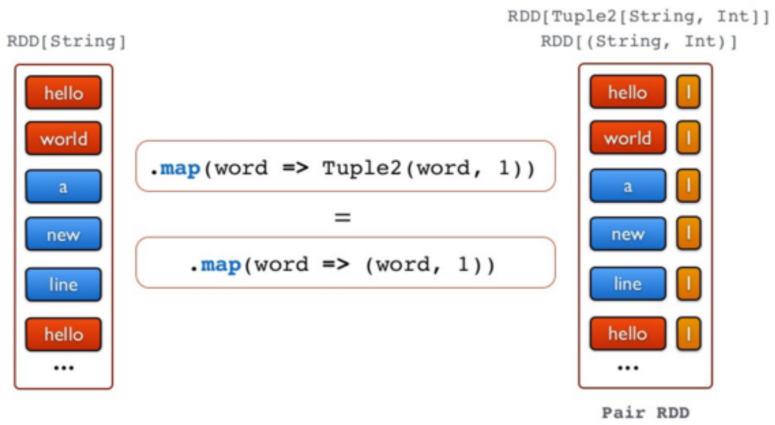

```
//Etape 2 - Convertir les lignes en minuscule
val lower = docs.map(line => line.toLowerCase)
```



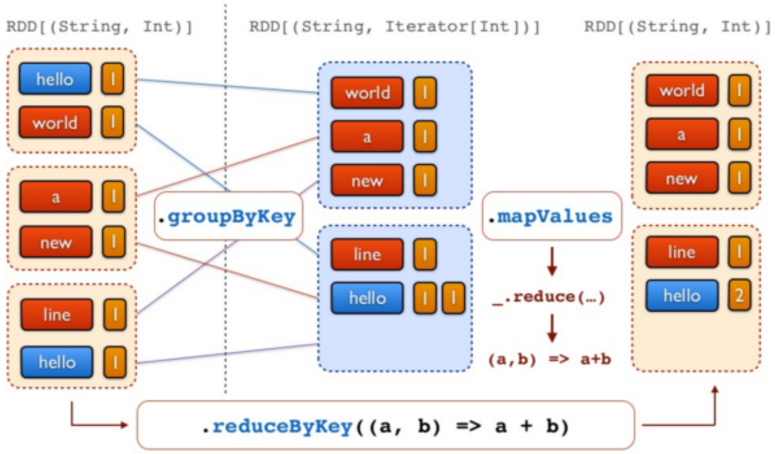

```
//Etape 3 - Séparer les lignes en mots
val words = lower.flatMap(line => line.split("\\s+"))
```



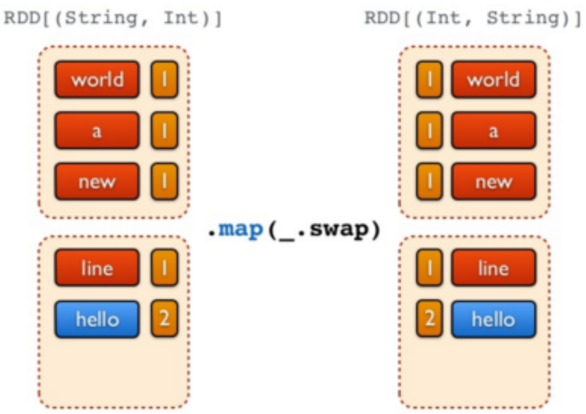

```
//Etape 4 - produire les tuples (mot, 1)
val counts = words.map(word => (word,1))
```



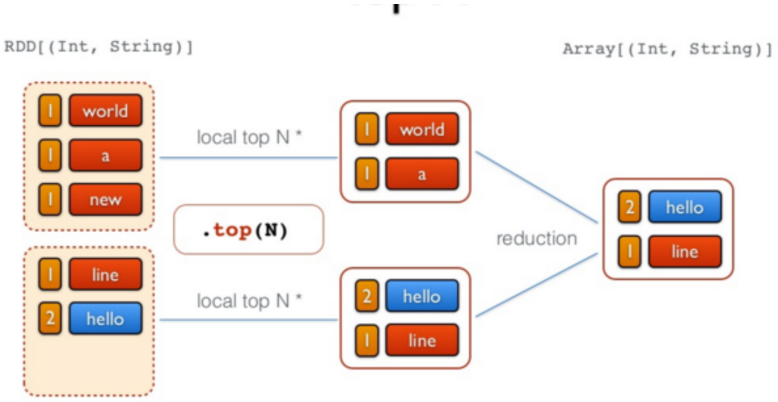

```
//Etape 5 - Compter tous les mots
val freq = counts.reduceByKey(_ + _)
```




```
//Etape 6 - Inverser les tuples (transformation avec swap)
freq.map(_.swap)
```




```
//Etape 6 - Inverser les tuples (action de sélection des 3 premiers)
val top = freq.map(_.swap).top(3)
```



Spark Batch en Java

Préparation de l'environnement et Code

Nous allons dans cette partie créer un projet Spark Batch en Java (un simple WordCount), le charger sur le cluster et lancer le job.

1. Créer un projet Maven avec VSCode, en utilisant la config suivante:

```
<groupId>spark.batch</groupId>
<artifactId>wordcount-spark</artifactId>
```

2. Rajouter dans le fichier pom les dépendances nécessaires, et indiquer la version du compilateur Java:

```
<properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>
<dependencies>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_2.13</artifactId>
        <version>3.5.0</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-reload4j</artifactId>
        <version>2.1.0-alpha1</version>
        <scope>test</scope>
```

```
    </dependency>
</dependencies>
```

3. Sous le répertoire java, créer un package que vous appellerez *spark.batch.tp21*, et dedans, une classe appelée *WordCountTask*.
4. Écrire le code suivant dans *WordCountTask.java* :

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import scala.Tuple2;

import java.util.Arrays;
import com.google.common.base.Preconditions;

public class WordCountTask {
      private static final Logger LOGGER = LoggerFactory.getLogger(WordCountTask.class);

      public static void main(String[] args) {
          Preconditions.checkArgument(args.length > 1, "Please provide the path of input file
and output dir as parameters.");
          new WordCountTask().run(args[0], args[1]);
      }

      public void run(String inputFilePath, String outputDir) {
          String master = "local[*]";
          SparkConf conf = new SparkConf()
                  .setAppName(WordCountTask.class.getName())
                  .setMaster(master);
          JavaSparkContext sc = new JavaSparkContext(conf);

          JavaRDD<String> textFile = sc.textFile(inputFilePath);
          JavaPairRDD<String, Integer> counts = textFile
                  .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
                  .mapToPair(word -> new Tuple2<>(word, 1))
                  .reduceByKey((a, b) -> a + b);
          counts.saveAsTextFile(outputDir);
      }
  }
```

La première chose à faire dans un programme Spark est de créer un objet *JavaSparkContext*, qui indique à Spark comment accéder à un cluster. Pour créer ce contexte, vous aurez besoin de construire un objet *SparkConf* qui contient toutes les informations sur l'application.

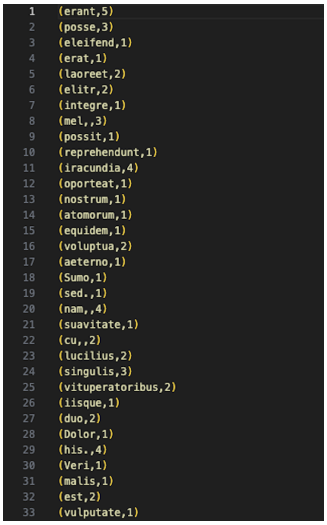
- *appName* est le nom de l'application
- *master* est une URL d'un cluster Spark, Mesos ou YARN, ou bien une chaîne spéciale *local* pour lancer le job en mode local.

> **Warning**
>
> Nous avons indiqué ici que notre master est *local* pour les besoins du test, mais plus tard, en le packageant pour le cluster, nous allons enlever cette indication. Il est en effet déconseillé de la hard-coder dans le programme, il faudrait plutôt l'indiquer comme option de commande à chaque fois que nous lançons le job.
>
> Le reste du code de l'application est la version en Java de l'exemple en scala que nous avions fait avec spark-shell.

Test du code en local

Pour tester le code sur votre machine, procéder aux étapes suivantes:

1. Insérer un fichier texte de votre choix (par exemple le fameux loremipsum.txt) dans le répertoire src/main/resources.
2. Lancer le programme en utilisant les arguments suivants:
    a. **Arg1**: le chemin du fichier *loremipsum.txt*
    b. **Arg2**: le chemin d'un répertoire *out* sous *resources* (vous ne devez pas le créer)
3. Cliquer sur OK, et lancer la configuration. Si tout se passe bien, un répertoire *out* sera créé sous *resources*, qui contient (entre autres) deux fichiers: part-00000, part-00001.



Lancement du code sur le cluster

Pour exécuter le code sur le cluster, modifier comme indiqué les lignes en jaune dans ce qui suit:

```
public class WordCountTask {
  private static final Logger LOGGER = LoggerFactory.getLogger(WordCountTask.class);

  public static void main(String[] args) {
      checkArgument(args.length > 1, "Please provide the path of input file and output dir as
parameters.");
      new WordCountTask().run(args[0], args[1]);
  }

  public void run(String inputFilePath, String outputDir) {

      SparkConf conf = new SparkConf()
              .setAppName(WordCountTask.class.getName());

      JavaSparkContext sc = new JavaSparkContext(conf);

      JavaRDD<String> textFile = sc.textFile(inputFilePath);
      JavaPairRDD<String, Integer> counts = textFile
              .flatMap(s -> Arrays.asList(s.split("\t")).iterator())
              .mapToPair(word -> new Tuple2<>(word, 1))
              .reduceByKey((a, b) -> a + b);
      counts.saveAsTextFile(outputDir);
  }
}
```

Lancer ensuite une configuration de type Maven, avec la commande *package*. Un fichier intitulé *wordcount-spark-1.0-SNAPSHOT.jar* sera créé sous le répertoire target.

Nous allons maintenant copier ce fichier dans docker. Pour cela, naviguer vers le répertoire du projet avec votre terminal (ou plus simplement utiliser le terminal dans VSCode), et taper la commande suivante:

```
docker cp target/wordcount-spark-1.0-SNAPSHOT.jar hadoop-master:/root/wordcount-spark.jar
```

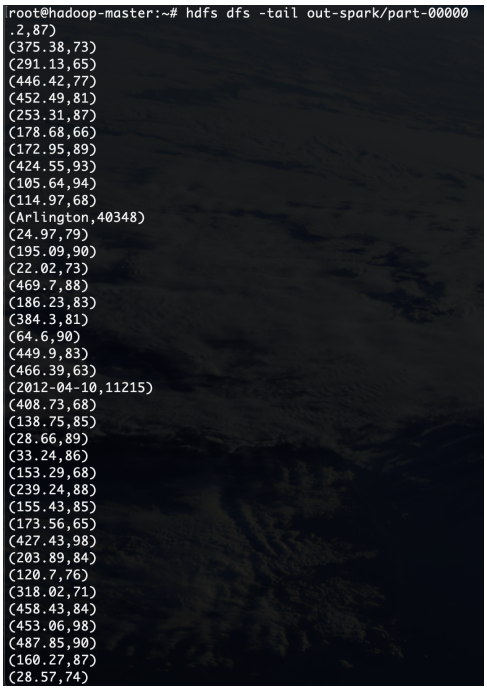
Revenir à votre contenaire master, et lancer un job Spark en utilisant ce fichier jar généré, avec la commande `spark-submit`, un script utilisé pour lancer des applications spark sur un cluster.

```
spark-submit  --class spark.batch.tp21.WordCountTask --master local wordcount-spark.jar
input/purchases.txt out-spark
```

- Nous allons lancer le job en mode local, pour commencer.
- Le fichier en entrée est le fichier purchases.txt (que vous déjà chargé dans HDFS dans le TP précédent), et le résultat sera stocké dans un nouveau répertoire *out-spark*.

> **Attention**
>
> Vérifiez bien que le fichier *purchases* existe dans le répertoire input de HDFS (et que le répertoire *out-spark* n'existe pas)! Si ce n'est pas le cas, vous pouvez le charger avec les commandes suivantes:
>
> ```
> hdfs dfs -mkdir -p input
> hdfs dfs -put purchases.txt input
> ```

Si tout se passe bien, vous devriez trouver, dans le répertoire *out-spark*, deux fichiers part-00000 et part-00001, qui ressemblent à ce qui suit:



Nous allons maintenant tester le comportement de *spark-submit* si on l'exécute en mode *cluster* sur YARN. Pour cela, exécuter le code suivant:

```
  spark-submit  --class spark.batch.tp21.WordCountTask --master yarn --deploy-mode cluster
wordcount-spark.jar input/purchases.txt out-spark2
```

- En lançant le job sur Yarn, deux modes de déploiement sont possibles:
  - **Mode cluster**: où tout le job s'exécute dans le cluster, c'est à dire les Spark Executors (qui exécutent les vraies tâches) et le Spark Driver (qui ordonnance les Executors). Ce dernier sera encapsulé dans un YARN Application Master.

- **Mode client** : où Spark Driver s'exécute sur la machine cliente (tel que votre propre ordinateur portable). Si votre machine s'éteint, le job s'arrête. Ce mode est approprié pour les jobs interactifs.

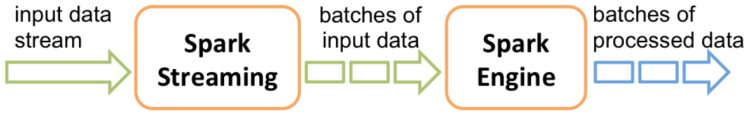
Si tout se passe bien, vous devriez obtenir un répertoire out-spark2 dans HDFS avec les fichiers usuels.

> **En cas d'erreur: consulter les logs!**
>
> En cas d'erreur ou d'interruption du job sur Yarn, vous pourrez consulter les fichiers logs pour chercher le message d'erreur (le message affiché sur la console n'est pas assez explicite). Pour cela, sur votre navigateur, aller à l'adresse: `http://localhost:8041/logs/userlogs` et suivez toujours les derniers liens jusqu'à *stderr*.

Spark Streaming

Spark est connu pour supporter également le traitement des données en streaming. Les données peuvent être lues à partir de plusieurs sources tel que Kafka, Flume, Kinesis ou des sockets TCP, et peuvent être traitées en utilisant des algorithmes complexes. Ensuite, les données traitées peuvent être stockées sur des systèmes de fichiers, des bases de données ou des dashboards. Il est même possible de réaliser des algorithmes de machine learning et de traitement de graphes sur les flux de données.

En interne, il fonctionne comme suit: Spark Streaming reçoit des données en streaming et les divise en micro-batches, qui sont ensuite calculés par le moteur de spark pour générer le flux final de résultats.



Environnement et Code

Nous allons commencer par tester le streaming en local, comme d'habitude. Pour cela:

1. Commencer par créer un nouveau projet Maven, avec le fichier pom suivant:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>spark.streaming</groupId>
  <artifactId>stream</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
      <maven.compiler.source>1.8</maven.compiler.source>
      <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
      <dependency>
          <groupId>org.apache.spark</groupId>
```

```
            <artifactId>spark-core_2.13</artifactId>
            <version>2.5.0</version>
        </dependency>
        <dependency>
          <groupId>org.apache.spark</groupId>
          <artifactId>spark-streaming_2.13</artifactId>
          <version>3.5.0</version>
      </dependency>

    </dependencies>
  </project>
```

2. Créer une classe *spark.streaming.tp22.Stream* avec le code suivant:

```
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Encoders;
import org.apache.spark.sql.SparkSession;
import org.apache.spark.sql.streaming.StreamingQuery;
import org.apache.spark.sql.streaming.StreamingQueryException;
import org.apache.spark.sql.streaming.Trigger;

import java.util.concurrent.TimeoutException;
import java.util.Arrays;

public class Stream {
    public static void main(String[] args) throws StreamingQueryException, TimeoutException  {
        SparkSession spark = SparkSession
            .builder()
            .appName("NetworkWordCount")
            .master("local[*]")
            .getOrCreate();

        // Create DataFrame representing the stream of input lines from connection to localhost:9999
        Dataset<String> lines = spark
            .readStream()
            .format("socket")
            .option("host", "localhost")
            .option("port", 9999)
            .load()
            .as(Encoders.STRING());

        // Split the lines into words
        Dataset<String> words = lines.flatMap(
            (String x) -> Arrays.asList(x.split(" ")).iterator(),
            Encoders.STRING());

        // Generate running word count
        Dataset<org.apache.spark.sql.Row> wordCounts = words.groupBy("value").count();

        // Start running the query that prints the running counts to the console
        StreamingQuery query = wordCounts.writeStream()
            .outputMode("complete")
            .format("console")
            .trigger(Trigger.ProcessingTime("1 second"))
            .start();

        query.awaitTermination();
    }
}
```

Ce code permet de calculer le nombre de mots dans un stream de données (provenant du port localhost:9999) chaque seconde. Dans sa version actuelle, Spark encourage l'utilisation de *Structured Streaming*,une API de haut niveau qui fournit un traitement plus efficace, et qui est construite au dessus de Spark SQL, en intégrant les structures DataFrame et Dataset.

Trigger Interval

Dans Spark Structured Streaming, le concept de microbatch est utilisé pour traiter les données en continu par petits lots incrémentaux. La durée de chaque micro-lot est configurable et détermine la fréquence de traitement des données en continu. Cette durée est appelée "intervalle de déclenchement". Si vous ne spécifiez pas explicitement d'intervalle de déclenchement, le trigger par défaut est *ProcessingTime(0)*, qui est aussi connu comme le mode de traitement par micro-lots. Ce paramètre par défaut signifie que Spark essaiera de traiter les données aussi rapidement que possible, sans délai fixe entre les micro-lots.

Test du code en Local

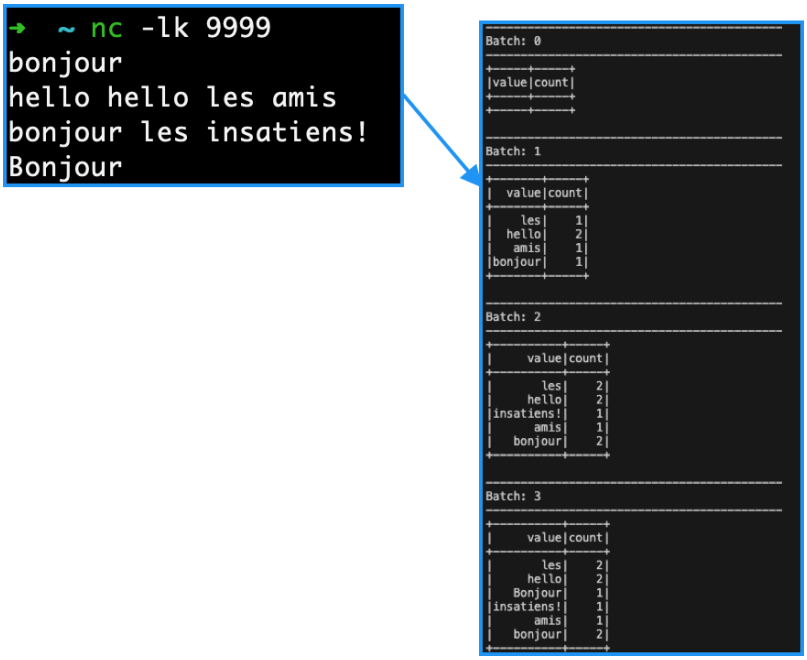
Le stream ici sera diffusé par une petite commande utilitaire qui se trouve dans la majorité des systèmes Unix-like.

- Ouvrir un terminal, et taper la commande suivante pour créer le stream:

```
nc -lk 9999
```

- Exécuter votre classe *Stream*. L'application est en écoute sur localhost:9999.
- Commencer à écrire des messages sur la console de votre terminal (là où vous avez lancé la commande nc)

A chaque fois que vous entrez quelque chose sur le terminal, l'application Stream l'intercepte, et l'affichage sur l'écran de la console change, comme suit:



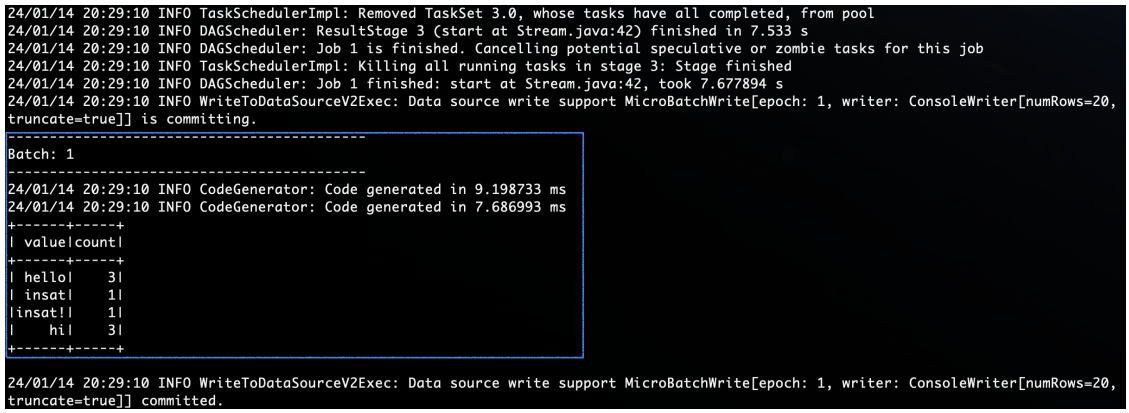
Lancement du code sur le cluster

Pour lancer le code précédent sur le cluster, il faudra d'abord faire une petite modification: **changer la valeur *localhost* par l'IP de votre machine hote** (celle que vous utilisez pour lancer la commande *nc*).

- Générer le fichier jar.
- Copier le fichier jar sur le contenaire master.
- Assurez-vous que la commande *nc* tourne bien sur votre machine, en attente de messages.
- Sur votre noeud master, lancer la commande suivante:

```
spark-submit  --class spark.streaming.tp22.Stream --master local stream.jar
```

Cette fois, énormément de texte est généré en continu sur la console. Comme nous avons défini dans l'application *console* comme sortie, le résultat du traitement s'affichera au milieu de tout ce texte. Une fois que vous aurez saisi le texte à tester, arrêter l'application (avec Ctrl-C), et chercher dans le texte la chaîne "**Batch:**". Vous trouverez normalement un résultat semblable au suivant:



Homework

Vous allez maintenant appliquer des traitements sur votre projet selon votre besoin. Vos contraintes ici est d'avoir les deux types de traitement: Batch et Streaming. Vous pouvez utiliser Map Reduce ou Spark pour le traitement en Batch.

Last update: 2023-02-09