

# TP: Cluster spark avec docker

Pour rappel, Spark est un moteur de calcul distribué destiné au traitement de données à grande échelle.

C'est un framework complet et unifié pour le traitements et l'analyse Big Data de diverse natures (texte, streaming, graphe, etc.).

Spark peut s'exécuter en mode standalone ou bien en mode cluster manager dédié tels qu' Apache Hadoop YARN ou Apache Mesos.

L'objectif de ce TP est de :

- ◆ Installer un cluster spark avec docker
- ◆ Premier exemple
- ◆ Installer pyspark sur colab
- ◆ charger et manipuler des données avec spark
- ◆ Étude de cas

## I. Installation Cluster Spark

### 1. Accéder au master

Après le démarrage du cluster hadoop, entrer dans le conteneur master

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

### 2. Démarrer hadoop et yarn

lancer hadoop et yarn en utilisant un script fourni appelé start-hadoop.sh.

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

```
./start-spark.sh
```

- A la fin du démarrage, vérifier si spark et yarn ont démarré correctement. Pour ce faire :
  - vérifier à travers la commande jps
  - Dans un navigateur, entrer l'adresse

Yarn Web UI <https://localhost:8088>

Spark web UI: <https://localhost:8080>

## II. Premiers exemples sur apache spark

### 1. Premier exemple spark avec spark-submit

**Spark-submit** est un script spark qui permet de lancer des applications sur un cluster. Il peut utiliser tous les cluster manager pris en charge par Spark via une interface uniforme.

- Tester l'exemple **SparkPi** qui permet de calculer la valeur PI. Pour se faire

```
spark-submit \  
--class org.apache.spark.examples.SparkPi \  

```

```
--master local[*] \
$SPARK_HOME/examples/jars/spark-examples_{version}.jar \
100
```

- rapporter le nombre PI calculé pour différentes valeurs

## 2. Application word count avec spark-shell

Les shells de Spark fournit un moyen simple et puissant pour analyser les données de manière interactive.

On veut réaliser l'exemple wordcount en scala. Pour se faire, ouvrir spark-shell. les objets **sc** représentant sparkContext et **spark** pour SparkSession sont fournis.

- Écrire le script scala

```
val data=sc.textFile("hdfs://hadoop-master:9000/user/root/input/alice.txt")
val count= data.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_+_ )
count.saveAsTextFile("hdfs://hadoop-master:9000/user/root/output/respark1")
```

## 3. Soumettre une Application python

- Ecrire le script python

```
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("yarn").appName('wordcount').getOrCreate()
data = spark.sparkContext.textFile("hdfs://hadoop-master:9000/user/root/input/alice.txt")
words=data.flatMap(lambda line: line.split(" "))
wordCounts = words.map(lambda word: (word, 1)).reduceByKey(lambda a,b:a +b)
wordCounts.saveAsTextFile("hdfs://hadoop-master:9000/user/root/output/rr2")
```

- Soumettre le script python via spark-submit
- Consulter les résultats enregistrés dans HDFS.
- Sortir de bash de hadoop-master
- Arrêter les trois conteneurs

```
docker stop hadoop-master hadoop-slave1 hadoop-slave2
```

## ■ Installer pyspark sur colab

Pyspark est une api Python pour spark. Qui permet de manipuler les RDD et dataframes. Google Colab ne fournit pas Spark par défaut, nous devons donc l'installer.

### ◆ Installer Apache Spark et PySpark

- Commencer d'abord par Ouvrir un nouveau notebook colab.
- Exécutez les commandes suivantes dans une cellule Colab :

```
!sudo apt update
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
#Check this site for the latest download link
https://www.apache.org/dyn/closer.lua/spark/spark-3.2.1/spark-3.2.1-
bin-hadoop3.2.tgz
!wget -q https://dlcdn.apache.org/spark/spark-3.2.1/spark-3.2.1-bin-
hadoop3.2.tgz
```

```
!tar xf spark-3.2.1-bin-hadoop3.2.tgz
!pip install -q findspark
!pip install pyspark
!pip install py4j
!pip install -q pymongo matplotlib seaborn
```

### ◆ Configurer l'environnement

Après l'installation, nous devons configurer les variables d'environnement pour utiliser Spark

```
import os
import sys
import findspark
# os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
# os.environ["SPARK_HOME"] = "/content/spark-3.2.1-bin-hadoop3.2"
findspark.init()
findspark.find()
```

### ◆ Démarrer une session Spark

```
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .appName("ColabSpark") \
    .config("spark.driver.memory", "2g") \
    .getOrCreate()
print(" Spark est configuré avec succès !")
```

## ■ Premier exemple

- Vérifier la session Spark

```
print(spark.version)
```

- Créer un DataFrame simple avec les infos suivantes

```
data = [(1, "Alice", 23), (2, "Bob", 30), (3, "Charlie", 29)]
columns = ["id", "nom", "age"]
df = spark.createDataFrame(data, columns)
```

- Afficher le contenu du DataFrame

```
df.show()
```

- voici Quelques opérations de base

```
df.printSchema() # Structure du DataFrame
df.select("nom", "age").show() # Sélection de colonnes
df.filter(df.age > 25).show() # Filtrage des données
```

## ■ Chargement et Manipulation des Données avec Spark

- Nous allons utiliser un jeu de données sur les transactions financières.

### ■ Charger un fichier CSV

1. On va charger un dataset CSV et l'analyser avec Spark

```
df = spark.read.csv("/content/transactions.csv", header=True, inferSchema=True)
df.show(5)
```

- **Afficher le schéma des données**

```
df.printSchema()
```

- **Filtrer les transactions supérieures à 1000**

```
df.filter(df["Transaction Amount"] > 1000).show()
```

- **Calculer le montant total des transactions par type**

```
df.groupBy("Transaction Type").sum("Transaction Amount").show()
```

- **Trier les transactions par montant décroissant**

```
df.orderBy(df["Transaction Amount"].desc()).show(5)
```

## ▪ Étude de cas : Intégration de Spark avec MongoDB Atlas

Nous allons maintenant connecter Spark à MongoDB Atlas pour analyser les transactions directement depuis la base de données.

- **Installer le connecteur MongoDB Spark**

```
!pip install pymongo
```

- **Configurer la connexion à MongoDB Atlas**

Ajoutez vos **identifiants MongoDB Atlas** :

```
mongo_uri =
"mongodb+srv://<username>:<password>@cluster0.mongodb.net/bankdb.transactions?
retryWrites=true&w=majority"
spark = SparkSession.builder \
    .appName("MongoDBIntegration") \
    .config("spark.mongodb.input.uri", mongo_uri) \
    .config("spark.mongodb.output.uri", mongo_uri) \
    .getOrCreate()
```

- **Charger les transactions depuis MongoDB**

Ajoutez vos **identifiants MongoDB Atlas** :

```
df_mongo = spark.read.format("mongo").option("uri", mongo_uri).load()
df_mongo.show(5)
```

- **Effectuer des analyses sur MongoDB avec Spark**
  - Calculer le montant moyen des transactions

```
df_mongo.groupBy("Transaction Type").avg("Transaction Amount").show()
```

- Trouver les comptes ayant effectué plus de 5 transactions

```
df_mongo.groupBy("Sender Account ID").count().filter("count > 5").show()
```

## • Utiliser Spark SQL

Spark permet aussi d'écrire des **requêtes SQL** sur les DataFrames.

- Enregistrer un DataFrame comme table temporaire

```
df.createOrReplaceTempView("transactions")
```

- Exécuter une requête SQL avec Spark

```
df.createOrReplaceTempView("transactions")
result = spark.sql("SELECT `Transaction Type`, SUM(`Transaction Amount`) as Total FROM
transactions GROUP BY `Transaction Type`")
result.show()
```

## • Visualiser les Transactions par Type

Nous allons afficher le **total des transactions par type** sous forme de graphique **barplot** avec **Seaborn**.

- Préparer les données

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
# Agréger les transactions par type
df_grouped = df.groupBy("Transaction Type").sum("Transaction Amount").toPandas()
# Renommer les colonnes
df_grouped.columns = ["Transaction Type", "Total Amount"]
df_grouped.sort_values(by="Total Amount", ascending=False, inplace=True)
```

- Visualiser avec Seaborn

```
plt.figure(figsize=(10, 5))
sns.barplot(data=df_grouped, x="Transaction Type", y="Total Amount",
palette="coolwarm")
plt.title("Montant Total des Transactions par Type")
plt.xlabel("Type de Transaction")
plt.ylabel("Montant Total (€)")
plt.xticks(rotation=45)
plt.show()
```

- Distribution des Montants des Transactions

Nous allons afficher la **distribution des montants** avec un **histogramme**.

- Convertir en Pandas

```
df_pandas = df.select("Transaction Amount").toPandas()
```

- Tracer un Histogramme

```
plt.figure(figsize=(10, 5))
```

```
sns.histplot(df_pandas["Transaction Amount"], bins=30, kde=True, color="blue")
plt.title("Distribution des Montants des Transactions")
plt.xlabel("Montant (€)")
plt.ylabel("Nombre de Transactions")
plt.show()
```

### I. Comparaison des Transactions Réussies vs Échouées

Nous allons comparer les **transactions réussies et échouées** avec un **countplot**.

#### ■ Préparer les données

```
df_status = df.groupby("Transaction Status").count().toPandas()
df_status.columns = ["Transaction Status", "Count"]
```

#### ■ Tracer le graphique

```
plt.figure(figsize=(7, 5))
sns.barplot(data=df_status, x="Transaction Status", y="Count", palette="pastel")
plt.title("Nombre de Transactions Réussies vs Échouées")
plt.xlabel("Statut de la Transaction")
plt.ylabel("Nombre de Transactions")
plt.show()
```