TP Spark : WordCount avec PySpark

Cluster Docker 3 Nœuds

Objectifs

- Comprendre les RDD : création, transformations, actions.
- Soumettre un job PySpark à un cluster Spark.
- Manipuler les partitions et produire un fichier de sortie unique.
- Déployer un cluster Spark via Docker Compose.

1 C'est quoi PySpark ?

PySpark est une bibliothèque Python fournissant une interface pour Apache Spark, un framework de traitement distribué et open source.

Spark est écrit en Scala, tandis que PySpark permet d'utiliser Python pour manipuler des RDD ou des DataFrames, appliquer des transformations et effectuer des analyses avancées. PySpark permet de traiter des données provenant de diverses sources (fichiers texte, HDFS, bases de données) et est couramment utilisé pour les pipelines ETL, les analyses massives ou le machine learning via MLlib.

2 Mise en place de l'environnement

Dans cette première étape, nous allons préparer notre environnement de travail en construisant l'image Docker qui contient Spark et PySpark. L'objectif est d'obtenir un mini–cluster Spark fonctionnel (1 Master et 2 Workers) que nous pourrons utiliser pour exécuter notre application WordCount.

2.1 Construction de l'image Docker Spark/PySpark

Le dossier `Requirements` mis à disposition, sur arche, contient déjà tout le nécessaire :

- un `Dockerfile` permettant de créer une image Spark intégrant PySpark,
- un `docker-compose.yml` décrivant un cluster Spark composé d'un Master et de deux Workers.

Nous commençons donc par construire l'image Docker. Pour cela, il faut d'abord se placer dans le dossier `Requirements` (placé dans votre répertoire `HOME`), car il contient le `Dockerfile` nécessaire à la construction de l'image. Ensuite via la ligne de commande exécuter les commandes suivantes :

```
docker build -t apache-spark:3.4.0 .
docker images
```

L'image générée repose sur le script `start-spark.sh` présent dans le dossier. Ce script sert de point d'entrée et adapte automatiquement le comportement du conteneur (selon la variable d'environnement $SPARK\_WORKLOAD$) pour le lancer en mode :

- Master Spark,
- Worker Spark,
- ou Submit (soumission d'un job).

Cette approche nous permet d'utiliser la même image pour tous les rôles du cluster.

2.2 Démarrage du cluster Spark

Une fois l'image construite, nous déployons le cluster Spark défini dans `docker-compose.yml` :

```
docker-compose up -d
```

Le fichier `docker-compose.yml` crée automatiquement :

- un conteneur **Master** responsable de la coordination,
- deux conteneurs **Workers** chargés d'exécuter les tâches.

Chaque conteneur monte deux répertoires locaux :

- `/app` : destiné aux scripts PySpark (ex. `wordcount.py`),
- `/data` : contenant les fichiers d'entrée et les résultats produits.

Ces volumes sont partagés entre tous les nœuds du cluster, garantissant que les scripts et les données sont accessibles de manière identique sur le Master et les Workers.

Pour vérifier que le cluster est correctement démarré :

```
docker ps
```

3 Mise en place d'une application Spark : WordCount

3.1 Analyse du script wordcount.py

Dans le dossier `Requirements`, le script `wordcount.py` contient différentes étapes PySpark :

- création du SparkContext,
- chargement d'un fichier texte,
- application de transformations RDD,
- action finale pour obtenir les résultats.

Copiez le script dans `/app/` et le fichier `test.txt` dans `/data/`.

3.2 Accéder au conteneur Master

```
docker exec -it spark-master /bin/bash
```

3.3 Soumission d'un job Spark

```
./bin/spark-submit \
  --master spark://spark-master:7077 \
  --name WordCount \
  /opt/spark-apps/wordcount.py
```

Explication de la commande

- `spark-submit` : outil pour exécuter des applications Spark.
- `-master spark://spark-master:7077` : adresse du Master Standalone.
- `-name WordCount` : nom visible dans l'interface web Spark.
- `/opt/spark-apps/wordcount.py` : chemin du script PySpark.

Il est important de rappeler que dans notre configuration, nous utilisons Spark en **mode Standalone**. Dans ce mode, le **Spark Master joue directement le rôle de Cluster Manager**.

Autrement dit, c'est le Master Standalone qui :

- reçoit l'application soumise via `spark-submit`,
- alloue des Executors sur les Workers disponibles,
- distribue les tâches à exécuter,
- et supervise l'exécution du job.

Dans notre cas, le gestionnaire de cluster utilisé est donc le **Standalone Manager**, et c'est le **Spark Master** qui assure entièrement ce rôle.

4 Résultat du job

Les fichiers de sortie sont produits dans :

```
/data/output/
```

Vous pouvez monitorer l'exécution sur :

- Interface Master : `http://localhost:8080`