TP2: Création d'un cluster Spark Standalone

- ❑ Installation et configuration Spark sous Linux
- ❑ Création d'un cluster mono-nœud
- ❑ Mise en place d'un cluster Spark standalone avec Docker

TP2: Installer Spark sous Linux (conteneur Docker) (1/3)

1. Installer Ubuntu via la console Windows (cmd)
   - ➢ **docker pull *ubuntu***
- ➢ **docker run -itd -p** 8080:8080 **-p** 7077:7077 **-p** 4040:4040 **--name spark1 --hostname spark1 ubuntu**
   - ➢ **docker ps**
   - ➢ **docker exec -it spark1 bash**

     Accéder au conteneur Spark1 via un terminal

     En cas de problème: **docker ps -a**

     docker start spark1
2. Installation Java
   - ➢ **apt update**
   - ➢ **apt -y upgrade**
   - ➢ **apt install default-jdk**
   - ➢ **java -version**
3. Installation scala: **apt install scala**

TP2: Installer Spark sous Linux (container Docker) (2/3)

4. Installation **Spark** dans le conteneur **Spark1:**
   - **apt install** curl
   - **curl -O** ***https://archive.apache.org/dist/spark/spark-3.5.0/spark-3.5.0-bin-hadoop3-scala2.13.tgz***
   - **tar xvf** ***spark-3.5.0-bin-hadoop3-scala2.13.tgz***
   - **mv spark-3.5.0-bin-hadoop3-scala2.13 /opt/spark**
   - **rm spark-3.5.0-bin-hadoop3-scala2.13.tgz**
5. Mise en place de l'environnement **Spark** en modifiant le ficheir **~/.bashrc** avc un éditeur de texte **(**Pour installer l'éditeur **vim**: **apt install vim)**
   - **vim ~/.bashrc**

     Ajouter à la fin du fichier les deux lignes suivantes:

     export SPARK_HOME=/opt/spark

     export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
   - Enregistrer et quitter **vim**: Appuyer sur **Echap** et taper **:wq**
   - **source ~/.bashrc**

TP2: Installer Spark sous Linux (container Docker) (3/3)

6. Créer une fichier texte **/home/data/test.txt** contenant quelques lignes
7. Lancer **spark-Shell** en local : *root@spark1:/#* **spark-shell**
8. Tester l'exemple **Scala** suivant dans **spark-shell** pour calculer le nombre d'occurrences de chaque mot dans le fichier **/home/data/test.txt** qu'il faut créer:

```
val lignes = sc.textFile("/home/data/test.txt")
val mots = lignes.flatMap(_.split("\\s+"))
val wc = mots.map(w => (w , 1)).reduceByKey(_+_)
wc.saveAsTextFile("/home/resultat")
```

9. Depuis le système **hôte (Windows)**, accéder à l'URL **http://localhost:4040** pour le monitoring de votre application (**Spark-shell**) lancée localement (**master = local[*]**)
10. Quitter **spark-shell** avec **:q**

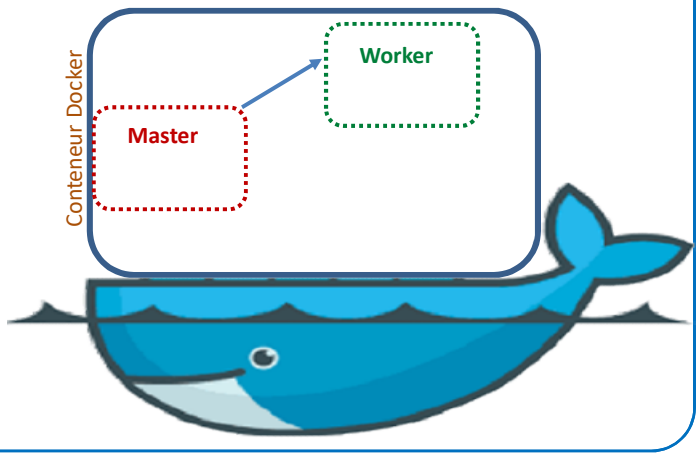
TP2: Créer un cluster Spark Standalone Mono-Noeud (1/4)

Objectif:

On veut créer un **cluster Spark** constitué d'**un seul noeud** qui est un conteneur Docker **spark1** crée dans la première partie de ce **TP** et sur lequel on lancera:

- Le **Spark Master** qui est le **maitre** du **gestionnaire de ressources intégré** de Spark.
- Un **Worker**

- Ce noeud va jouer le rôle du **Maître** et d'un **Worker**.
- Les **deux processus** qui leur sont associés seront tous les deux **lancés dans le conteneur**.



TP2: Créer un cluster Spark Standalone Mono-Noeud (2/4)

1. Dans le conteneur **Spark1**, lancer le **Master** du **cluster Spark standalone**: **start-master.sh**

   L'URL du Spark Master est: spark://spark1:7077

2. Démarrer ensuite un **Worker** dans le même conteneur **Spark1**:

   start-slave.sh spark://spark1:7077 OU start-worker.sh spark://spark1:7077

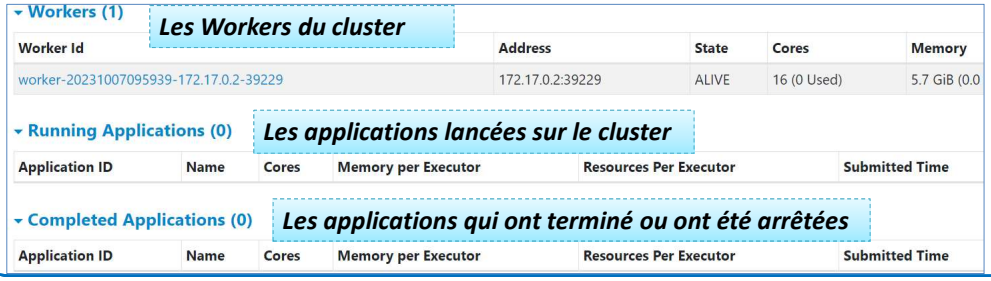
   Remarque : start-slave est déprécié

   - lancer la commande **jps** pour afficher les processus **Java** en cours.

   Le Spark Master

3. utilisez 'adresse suivante pour le monitoring du **cluster Spark standalone mono-nœud** :

   http://localhost:8080



TP2: Créer un cluster Spark Standalone Mono-Noeud (3/4)

4. Lancer l'application **spark-shell** dans le conteneur **Spark1** et lui allouer **3 cœurs** CPU virtuels:

   spark-shell --master spark://spark1:7077 --total-executor-cores 3

5. Actualiser la page du monitoring du **cluster Spark** (**http://localhost:8080**) pour vérifier si l'application **spark-shell** en **cours d'exécution** y apparaît:

Application ID		Name	Cores	Memory per Executor
app-20231007111841-0002	(kill)	Spark shell	3	1024.0 MiB

6. Utiliser un autre terminal du conteneur **Spark1** pour lancer le shell **pyspark** et lui allouer **3 cœurs** CPU virtuels : **pyspark --master spark://spark1:7077 --total-executor-cores 3**

> **Si on n'indique pas le nombre de cœurs, l'application prendra tous ceux qui sont disponibles sur le cluster.**

7. Actualiser la page **http://localhost:8080** pour avoir maintenant deux applications en cours d'exécution

Application ID		Name	Cores	Memory per Executor
app-20231007111900-0003	(kill)	PySparkShell	3	1024.0 MiB
app-20231007111841-0002	(kill)	Spark shell	3	1024.0 MiB

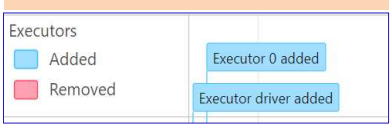
TP2: Créer un cluster Spark Standalone Mono-Nœud (4/4)

8. Tester l'exemple **Scala** suivant dans **spark-shell** pour calculer le nombre d'occurrences de chaque mot dans le fichier **/home/data/test.txt** :

```
val lignes = sc.textFile("/home/data/test.txt")
val mots = lignes.flatMap(_.split("\\s+"))
val wc = mots.map(w => (w , 1)).reduceByKey(_+_)
wc.saveAsTextFile("/home/resultat")
```

9. Appliquer la commande **ls** au dossier **/home/resultat/**
10. Afficher le contenu des fichiers résultat avec la commande: **cat /home/resultat/***
11. Depuis le système **hôte (Windows)**, accéder à l'URL suivant pour le monitoring de votre application (**Spark-shell**) lancée sur le **cluster Spark mono-nœud**: **http://localhost:4040**

Le **Driver** et un **Exécuteur**



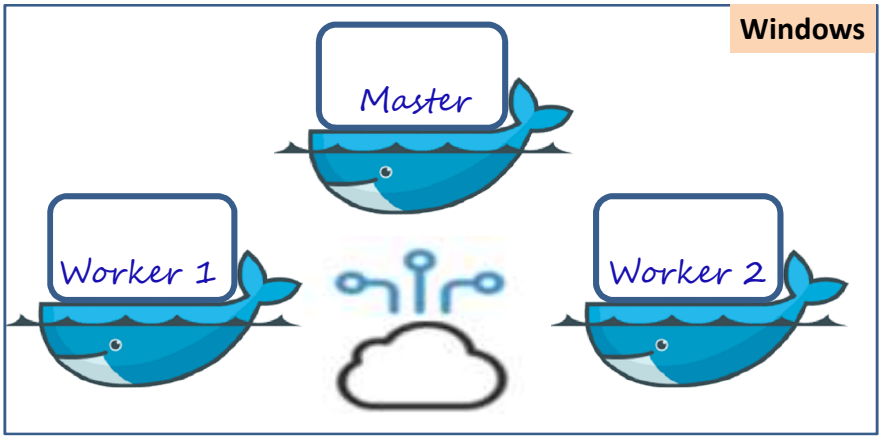
Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores
0	172.17.0.2:33599	Active	0	0.0 B / 434.4 MiB	0.0 B	3
driver	spark1:46561	Active	0	0.0 B / 434.4 MiB	0.0 B	0

L'exécuteur a les 3 cœurs alloués à l'application Spark-shell par ce qu'elle est déployée en mode **Client**: Le Driver n'est pas exécuté sur un **Worker** du cluster

TP2:Créer un cluster Spark Standalone de 3 nœuds (1/9)

Objectif:

- On veut créer un **cluster Spark** constitué de **3 noeuds** qui sont des **conteneurs Docker** dont le contenu est similaire à celui du conteneur **Spark1:** **Le Master et 2 Workers**
- Les **3** conteneurs seront intercconnectés via un réseau Docker



TP2:Créer un cluster Spark Standalone de 3 nœuds (2/9)

1. Installer les package net-tools et iputils-ping avec:
   - **apt install -y net-tools**
   - **apt install -y iputils-ping**
2. Installer **OpenSSH** sur la machine
   - **apt install openssh-server openssh-client**
3. Générer une paire de clefs (***valider le chemin par défaut proposé***)
   - **ssh-keygen -t rsa -P ""**
4. Définir la clef générée comme clef autorisée
   - **cp /root/.ssh/id_rsa.pub /root/.ssh/authorized_keys**
5. Ajouter la ligne suivante à la fin du fichier **~/.bashrc** pour lancer **ssh** au démarrage du conteneur:
   - **service ssh start**

TP2:Créer un cluster Spark Standalone de 3 nœuds (3/9)

6. Créer une copie du template du fichier **spark-env.sh** et le renommer:

   cp $SPARK_HOME/conf/spark-env.sh.template $SPARK_HOME/conf/spark-env.sh

7. Ajouter à la fin du fichier **~/.bashrc** : **export SPARK_WORKER_CORES=8**
8. Recharger ce fichier après modification avec: **source ~/.bashrc**
9. Créer le fichier de configuration **slaves** dans le répertoire **$SPARK_HOME/conf** et y ajouter les noms des conteneurs **workers** qui seront crées par la suite:

   - **vim $SPARK_HOME/conf/slaves**

   Les noms sont sur deux lignes: **spark-worker1**

   spark-worker2

10. On va créer maintenant une **image Docker** à partir du conteneur qu'on vient de configurer. Cette image va nous permettre de créer les trois conteneurs (nœuds) du cluster. Dans une **console Windows**, taper la commande suivante:

    - **docker commit spark1 spark_image**

TP2:Créer un cluster Spark Standalone de 3 nœuds (4/9)

11. Taper la commande suivante pour vérifier que spark_image existe: **docker images**
12. Créer un réseau qui permettra de connecter les trois noeuds du cluster

    - **docker network create --driver=**bridge *spark_network*

13. Créer et lancer le conteneur du **Master** (le ***port 22 pour SSH***):

    - **docker run -itd --net=***spark_network* **-p** 8080:8080 **--expose** 22 **--name** *spark-master* **--hostname** *spark-master* ***spark_image***

14. Créer et lancer le conteneur du premier **Worker**:

    - **docker run -itd --net=***spark_network* **--expose** 22 **--name** *spark-worker1* **--hostname** *spark-worker1* ***spark_image***

TP2:Créer un cluster Spark Standalone de 3 nœuds (5/9)

15. Créer et lancer le conteneur du deuxième **Worker**:
    - **docker run -itd --net**=*spark_network* **--expose** 22 **--name** *spark-worker2* **--hostname** *spark-worker2* ***spark_image***
16. Vérifier que les trois conteneurs sont créés: **docker ps**
17. Obtenir l'adresse IP du conteneur **spark-master**:
    - Depuis une **console Windows**: **docker exec -it** spark-worker1 **bash**
    - Depuis le terminal du conteneur: **ifconfig**

    ➔ Noter l'adresse IP, par exemple: **172.18.0.2**
18. Accéder au terminal du conteneur **spark-worker1** et lancer le premier **Worker**:
    - **start-worker.sh** spark://**172.18.0.2**:7077
19. Faire de même pour le deuxième conteneur **spark-worker1**

TP2:Créer un cluster Spark Standalone de 3 nœuds (6/9)

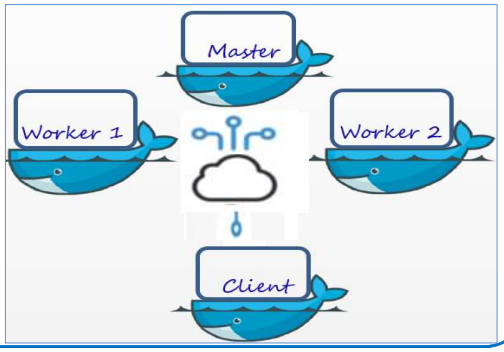
21. Accéder à la pge du monitoring du **cluster Spark standalone créé** : **http://localhost:8080**

Noter le nombre de **Workers** et les **ressources allouées** à chacun

▾ Workers (2)

Worker Id	Address	State	Cores	Memory
worker-20231011185531-172.18.0.3-45361	172.18.0.3:45361	ALIVE	8 (8 Used)	5.7 GiB (1024.0 MiB Used)
worker-20231011185701-172.18.0.4-38249	172.18.0.4:38249	ALIVE	8 (8 Used)	5.7 GiB (1024.0 MiB Used)

22. Créer et lancer un conteneur **spark-client** dans le même réseau **spark_network**:
    - **docker run -itd --net**=*spark_network* **-p** 4040:4040 **--expose** 22 **--name** *spark-client* **--hostname** *spark-client* ***spark_image***
23. Accéder au terminal du conteneur **spark-client** et lancer le shell Python:
    - **pyspak --master** spark://**172.18.0.2**:**7077**



TP2:Créer un cluster Spark Standalone de 3 nœuds (7/9)

24. Actualiser la page du monitoring du **cluster Spark standalone créé** :

    Noter le nombre d'applications lancées sur le cluster

▾ **Running Applications (1)**

Application ID		Name	Cores	Memory per Executor
app-20231011193953-0000	(kill)	PySparkShell	16	1024.0 MiB

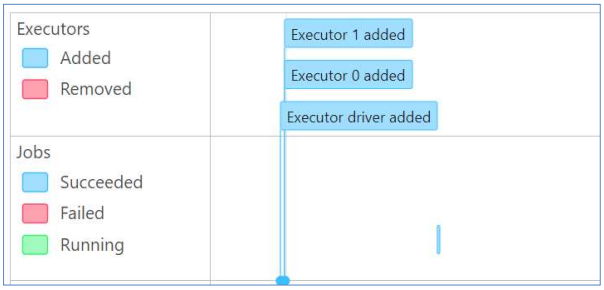
25. Dans le conteneur **spark-client**, exécuter le code suivant dans le shell Python:

```
>>> data = [('id1', '50'), ('id2', '30'), ('id1', '10'), ('id2', '20'), ('id1', '3'), ('id2', '7')]
>>> RDD1 = sc.parallelize(data)
>>> RDD2 = RDD1.groupByKey()
>>> for key, values in RDD2.collect(): print(f"(Client: {key}, Montants:{list(values)})")
```


TP2:Créer un cluster Spark Standalone de 3 nœuds (8/9)

26. Accéder à la page de monitoring de l'application pyspark lancée sur le conteneur spark-client:

    N.B: Il y a le **Driver** de l'application et **deux Executors** dont chacun est lancé sur un Worker.



TP2:Créer un cluster Spark Standalone de 3 nœuds (9/9)

27. Dans le conteneur **spark-client**, créer le fichier **Exemple.py** suivant:

```
# Créer un objet SparkContext
        sc = SparkContext("spark://172.18.0.2:7077", "Exemple ")
# Données à traiter
        data = [('id1', '50'), ('id2', '30'), ('id1', '10'), ('id2', '20'),  ('id1', '3'), ('id2', '7')]
# Crée un  RDD à partir du contenu de data
        RDD1 = sc.parallelize(data)
# Utiliser groupByKey pour regrouper les pairs de RDD1 par clé
        RDD2 = RDD1.groupByKey()
# Itérer sur les groupes et agir sur les valeurs
        for key, values in RDD2.collect(): print(f"(Client: {key}, Montants:{list(values)})")
```

28. Déployer cette application Python sur le cluster avec **Spark-Submit**:

➢ **spark-submit --master spark://172.18.0.2:7077 *exemple.py***

29. Actualiser la page du monitoring du **cluster Spark standalone créé .**

N.B: Noter le nombre d'application terminées et les ressources consommées.