TP Spark Java **Module IBD**

L'objectif de ce TP est d'expérimenter le framework de calcul distribué Spark vue en cours à travers son interface Java. Ce framework est destiné aux applications de BigData et d'IA en prenant en charge la répartition de données et des traitements dans un cluster de machines. Il a été avec le langage Scala (compatible avec la machine virtuelle Java) et s'appuie sur HDFS et YARN (couches logicielles de bas niveau fournies par Hadoop) pour le stockage et l'exécution.

Comme dans le cas du TP Hadoop, vous utiliserez le mode distribué en passant par des conteneurs Docker. L'utilisation des conteneurs Docker va garantir la consistance entre les environnements de développement (Linux, Window, Macos) et permettra de réduire considérablement la complexité de configuration des machines.

Partie 1: Installation, Configuration, Premier test

Commencez par installer une version de Docker sur votre machine.

Nous allons utiliser des scripts pour faciliter le travail de construction des images et le lancement des exécutables utiles à l'exécution de Spark. Pour cela, il faut cloner le projet git à l'adresse :
`https://github.com/gvanwormhoudt/docker-spark-cluster.git`

Ce projet est similaire à celui utilisé pour Hadoop. Il s'occupe de la configuration de Spark, en fournissant :
1) une image debian avec scala (image scalabase).
2) trois nœuds Spark entièrement configurés fonctionnant au dessus de HDFS et YARN (image sparkbase) :
* nodemaster (nœud maître)
* node2 (esclave)
* node3 (esclave)

Installation et Déploiement

Le fichier README.md contient les instructions pour construire les images dockers et réaliser le déploiement. Ces instructions sont redonnées ci-dessous :

1) Clonez le dépôt
2) `cd scalabase`
3) ./build.sh # Ceci construit le conteneur debian
4) `cd ../sparkbase`
5) `./build.sh` # Cela construit l'image sparkbase
6) exécuter `./cluster.sh deploy`
Cela aura pour effet de démarrer les conteneurs et de lancer les services HDFS, YARN et SPARK dans chacun d'eux
7) Le script finira d'afficher les URL d'administration de YARN et de HDFS:
* Hadoop info @ nodemaster : http://<adresse ip>:8088/cluster
* DFS Health @ nodemaster : http://<adresse ip>:9870/dfshealth.html
* Spark @ nodemaster: http://<adresse ip>:8080

En utilisant la commande docker ou Docker Deskop, vérifiez la bonne exécution des conteneurs. Vous pouvez également vérifier l'existence du réseau sparknet qui relie les conteneurs entre eux. Vous pouvez également vérifier l'exécution en ouvrant un navigateur web aux urls précédentes en remplaçant l'adresse IP par localhost.

Une fois déployé le cluster peut être arrêté et redémarré avec la commande ./cluster.sh suivi de stop ou start.

Gestion de fichiers HDFS et Exécution d'un job Spark

Entrer dans le conteneur master en tant qu'utilisateur « hadoop » pour commencer à utiliser HDFS.

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

Le résultat de cette exécution sera le suivant:

```
hadoop@nodemaster:~$
```

Vous vous retrouverez dans le shell du nodemaster et vous pourrez ainsi manipuler le cluster à votre guise. Dans le conteneur, HDFS et YARN sont déjà lancés (travail du script cluster.sh que vous pouvez analyser au passage).

Les instructions qui suivent permettent d'exécuter un Job Spark. Cette séquence sera à répéter pour le programme développé dans la partie suivante. Avant le lancement, il faut créer les fichiers d'entrée (En principe, le système de fichier est déjà formaté dans les images du conteneur mais c'est parfois une étape à réaliser). Ici, on prendra un simple fichier de texte (file.txt) en entrée qu'on copiera dans le dossier /home/hadoop du conteneur nodemaster.

- Création des répertoires HDFS requis pour exécuter les jobs Spark

```
$ hdfs dfs -mkdir /input
```

- Copie du fichier servant en entrée dans le répertoire créé précédemment (ici on copie les fichiers du répertoire local etc/hadoop

```
$ hdfs dfs -put $HADOOP_HOME/file.txt /input
```

- Exécution d'un exemple de Job fourni avec la plateforme (exemple calcul de PI)

```
$ spark-submit run-example --class org.apache.spark.examples.SparkPi
```

Partie 2: Programmation

Comme indiqué en cours, Spark propose plusieurs abstractions pour manipuler des grandes collections de données: *RDD*, *DataFrame* et *Dataset*. Dans le cadre de ce TP, nous utiliserons principalement le type *Dataframe* car les données sont structurées en tableau. Les données à utiliser sont les mêmes que celles utilisés pour le TP Hadoop mais dans une version tabulaire au format csv (voir le fichier big_access.csv). Après avoir récupéré ce fichier de MyLearningSpace vous pouvez le copier dans le répertoire /home/hadoop du conteneur nodemaster.

Afin d'être prêt plus rapidement pour coder, deux fichiers supplémentaires sont à récupérer dans MyLearningSpace. Il s'agit :

- du fichier LogsApp.java : squelette de programme principale qui contient les premières instructions pour créer les objets initiaux (le SparkConf et le Dataset) ainsi que des méthodes à remplir pour chaque étape
- du fichier pom.xml : spécification Maven (dépendances) pour permettre la compilation de programme Spark en Java

Après avoir télécharger ces fichiers, vous pouvez créer un projet Eclipse basé sur ces fichiers.

À partir de maintenant, vous allez pouvoir utiliser les opérateurs disponibles sur les objets *Dataframe* afin de répondre aux questions suivantes, relatives au jeu de données.

Pour chaque question, Il est demander d'écrire une ou plusieurs lignes de code, de l'exécuter et de vérifier le résultat. Une indication sur le ou les opérateurs à utiliser est donnée mais il est parfois possible d'opérer autrement. Les question sont structurées en plusieurs catégories pour faciliter la progression et la compréhension.

Inspection des données

- Ecrire du code permettant d'obtenir des informations sur les données comme le nom et le type des colonnes, une description des valeurs numériques (min, max, ...), etc. Les éléments à utiliser sur le Dataframe sont : propriété *columns*, méthodes *show()*, *printSchema()*, *describe()*

Comptage

- Compter le nombre d'entrée dans le jeu de donnée (utilisation de la méthode *count()* de *Dataframe*)

- Compter le nombre d'adresse IP unique dans le jeu de donnée (utilisation des méthodes *select()*, *distinct()* et *count()* de *Dataframe)*.

Filtrage

- Déterminer le nombre de requêtes avec succès et possédant une adresse IP avec le préfixe "178." (utilisation des méthodes filter() et show()).

Aggrégation

* Calculer la moyenne des données téléchargés après 2017 (utilisation des méthodes withColumn(), select(), year(), col(), filter(), avg())

Tri

- Trier les requêtes avec succès selon leur taille décroissante et afficher le résultat du tri en affichant uniquement l'adresse IP et la taille (utilisation des méthodes *filter()*, *sort(), select()* et *show()*).

Regroupement

- À l'aide de l'opération de regroupement, afficher le nombre moyen de données téléchargées par adresse IP (utilisation des méthodes *groupBy()*, *mean()* et *show()*)

Transformation

- Créer un nouveau *Dataframe* à partir du *Dataframe* d'origine muni d'une nouvelle colonne 'size kb' dont les valeurs sont calculées par division des valeurs contenus dans 'size' par 100 (utilisation de la méthode *withColumn()* et de la fonction *udf()*)

Ecriture

- Ecrire dans un fichier un *Dataframe* ne contenant que les requêtes en échec sur une période donnée (utilisation de *filter()* et de *write.save("result", format="csv")* sur le *Dataframe)*