TP Spark

Yahn Formanczak

Version v1.0, 2022-01-17

Sommaire

1. Introduction

Apache Spark est un framework open source de calcul distribué. Il s'agit d'un ensemble d'outils et de composants logiciels structurés selon une architecture définie. Développé à l'université de Californie à Berkeley par AMPLab, Spark est aujourd'hui un projet de la fondation Apache. Ce produit est un cadre applicatif de traitements big data pour effectuer des analyses complexes à grande échelle.

Spark réalise une lecture des données au niveau du cluster, effectue toutes les opérations d'analyse nécessaires, puis écrit les résultats à ce même niveau.

De ce fait, là où le MapReduce de Hadoop travaille par étape, Spark peut travailler sur la totalité des données en même temps. Il est donc jusqu'à dix fois plus rapide pour le traitement en lots et jusqu'à cent fois plus rapide pour effectuer l'analyse en mémoire.

Spark exécute la totalité des opérations d'analyse de données en mémoire et en temps réel. Il s'appuie sur des disques seulement lorsque sa mémoire n'est plus suffisante. À l'inverse, avec Hadoop les données sont écrites sur le disque après chacune des opérations. Ce travail en mémoire permet de réduire les temps de latence entre les traitements, ce qui explique une telle rapidité.

Cependant, Spark ne dispose pas de système de gestion de fichier qui lui est propre. Il est nécessaire de lui en fournir un, par exemple Hadoop Distributed File System, Informix, Cassandra, OpenStack Swift ou Amazon S3).

En cas de panne ou de défaillance du système : les objets de données sont stockés dans ce que l'on appelle des ensembles de données distribués résilients (RDD : Resilient Distributed Datasets) répartis sur le cluster de données permettant la récupération complète de données.

Un RDD est une collection de données calculée à partir d'une source et conservée en mémoire vive. L'un des avantages apportés par RDD se trouve dans sa capacité à conserver les informations sur la manière dont une partition RDD a été produite. En cas de perte d'une partition il est donc en

mesure de la recalculer.

Spark fournit par défaut des interfaces de programmation qui permettent de l'utiliser depuis les langages Scala, Java, Python, et R.

1.1. PySpark

PySpark

1.2. Installation

Commençons par installer Spark. Sous Ubuntu, suivre la procédure suivante :

```
cd /tmp/
curl -O https://dlcdn.apache.org/spark/spark-3.2.1/spark-3.2.1-bin-hadoop3.2.tgz
tar xvf spark-*
mv spark-3.2.1-bin-hadoop3.2 /opt/spark
rm spark-*
```

Ajoutons les nouvelles constantes au .profile pour la bonne exécution des programmes :

```
echo "export SPARK_HOME=/opt/spark" >> ~/.profile
echo "export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin" >> ~/.profile
echo "export PYSPARK_PYTHON=/usr/bin/python3" >> ~/.profile
```

Afin de travailler sous Python, nous allons utiliser les librairies `PySpark` et `sparkfind`.
Les installer avec la commande pip :

```
pip install pyspark
pip install findspark
```

Enfin, démarrons le noeud maître de Spark :

```
/opt/spark/sbin/start-master.sh
```

Ce qui rendra l'interface web disponible sur le port 8080 : http://localhost:8080/

Pour démarrer un worker, il est nécessaire de lui fournir l'URL du maître :

```
./sbin/start-worker.sh <master-spark-URL>
# Exemple : ./sbin/start-worker.sh spark://spark1:7077
```

Enfin, pour arrêter les noeuds, nous pouvons utiliser les commandes suivantes :

```
SPARK_HOME/sbin/stop-slave.sh
$ SPARK_HOME/sbin/stop-master.sh
```

2. Cas pratique

Afin de travailler avec un jeu de données conséquent, nous allons travailler avec le script d'une pièce de théatre.
Le projet Gutemberg sauvegarde sous différents formats des oeuvres du domaine public afin que tout le monde puisse y avoir accès gratuitement : https://www.gutenberg.org/
Nous utiliserons le texte en français de Cyrano de Bergerac :

https://www.gutenberg.org/files/1256/1256-0.txt

COQUELIN
dans le rôle de Cyrano de Bergerac.

Nous allons analyser le contenu de ce script grâce à un algorithme WordCount.

2.1. Préparation du jeu de données

Voici les étapes à réaliser :

- Créez un script `SaveScript.py`
- Téléchargez le fichier texte en local grâce à la librairie `requests`
- Nettoyez le texte en enlevant les licences Gutenberg Project au début et à la fin du document grâce à la librairie `re`

```
import os
import requests
import re

URL_SRC='https://www.gutenberg.org/files/1256/1256-0.txt'
WORKDIR='/tmp/scripts/'

def get_script(url):
    # TODO

def clean_script(text):
    # TODO
    return text


if __name__ == "__main__":
    get_script(URL_SRC)
```

2.2. Prise en main de PySpark

Créons un nouveau script `WordCountScript.py`

Réalisons les imports nécessaires et l'initialisation de la session Spark

```
import findspark
from pyspark.sql import SparkSession
from pyspark.conf import SparkConf

findspark.init()

spark=SparkSession.builder\
    .master("local[*]")\
    .appName("WordCountScript")\
    .getOrCreate()
sc=spark.sparkContext
```

Puis chargeons le fichier :

```
# Chargement du fichier
script_rdd = sc.textFile(input_file)
```

Ajoutons une fonction de nettoyage du fichier :

```
# Nettoyage des caracteres speciaux
def lower_clean_str(x):
  punc='!"#$%&\'()*+,./:;<=>?@[\\]^_`{|}~-'
  lowercased_str = x.lower()
  for ch in punc:
    lowercased_str = lowercased_str.replace(ch, '')
  return lowercased_str
```

Que nous appellerons sur le RDD :

```
# Nettoyage
script_rdd = script_rdd.map(lower_clean_str)
```

Séparons chaque ligne du fichier en différents mots :

```
# Separation par mot
script_rdd = script_rdd.flatMap(lambda line: line.split(" "))
```

Enlevons les lignes vides :

```
# Suppression des mots vides
script_rdd = script_rdd.filter(lambda x:x!='')
```

Créons un système de clé valeur :

```
script_count=script_rdd.map(lambda  word:(word,1))
```

Réduction des données :

```
# Reduce By Key
script_count_RDK=script_count.reduceByKey(lambda x,y:(x+y)).sortByKey()
```

Et affichons le résultat :

```
script_count_RBK.take(40)
```