TP 2: Big Data - MapReduce

Partie 1 :

Le but de cette partie est de tester l'exemple du cours wordCount sur le cluster avec MapReduce. Pour ce faire, on va reprendre l'environnement Hadoop crée en Lab 0.

Etape 1 : Test de MapReduce en version Java sur un cluster Hadoop

La version Java de Map Reduce pour wordCount est déjà installée par défaut avec hadoop. Il faut juste installer le fichier input sur HDFS dans un dossier qu'on nommera input. Le dossier de sortie qu'on nommera output sera créé automatiquement sur HDFS et la sortie du programme sera un fichier stocké dans output.

Créer sur hdfs le dossier input et mettre dedans les fichiers qui serviront d'entrée à MapReduce (cf. Atelier HDFS) .

```
hadoop fs -mkdir -p input

hadoop fs -put /data/file1.txt input

hadoop fs -ls input
```

Pour exécuter MapReduce wordCount en java

```
hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/sources/hadoop-mapreduce-examples-3.3.6-sources.jar org.apache.hadoop.examples.WordCount input output
```

Vérifier le contenu du dossier de sortie output dans hdfs

```
hadoop fs -ls output

hadoop fs -cat output/part-00000
```

Etape 2 : Écriture des programmes Map et Reduce et test en local

Créer un sous-répertoire MapReduce et mettre les fichiers mapper.py et reducer.py.

On rappelle que L'API Hadoop Streaming est utilisée pour transmettre des données entre le code Map et Reduce via STDIN et STDOUT. Pour lire les données d'entrée et imprimer la sortie, "sys.stdin" est utilisé. D'autres procédures sont gérées par le streaming Hadoop lui-même.

On rappelle le contenu des programmes mapper.py et reducer.py. Une explication du code peut se trouver dans https://linuxhint.com/mapreduce-framework-python/ par exemple.

Notez qu'on va utiliser python3 et donc il faut en tenir compte dans le mapper.py et reducer.py

---

```
#!/usr/bin/python3

"""mapper.py"""

import sys

# input comes from STDIN (standard input)

for line in sys.stdin:

        # remove leading and trailing whitespace

        line = line.strip()

        # split the line into words

        words = line.split()

        # increase counters

        for word in words:

                # write the results to STDOUT (standard output);

                # what we output here will be the input for the

                # Reduce step, i.e. the input for reducer.py

                # tab-delimited; the trivial word count is 1

                print ('%s\t%s' % (word, 1))# create input directory on HDFS
```

```
#!/usr/bin/python3

"""reducer.py"""

import sys

current_word = None
current_count = 0
word = None

# input comes from STDIN

for line in sys.stdin:

    # remove leading and trailing whitespace
    line = line.strip()
    # splitting the data on the basis of tab we have provided in
mapper.py
    word, count = line.split('\t', 1)
```

```
    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word

# do not forget to output the last word if needed!
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

Il faut tester en local ces programmes avant de passer sur le cluster. Rendre les fichiers Mapper.py et Reducer.py exécutables si ce n'est pas déjà fait

```
chmod +x data/MapReduce/*
```

Sur votre terminal :

```
cat file.txt | python3 mapper.py | sort -k1,1 | python3 reducer.py
```

Si tout va bien, on passe au cluster Hadoop.

Etape 3 : Test de MapReduce en version python sur un cluster Hadoop

Tester votre code python avec la commande suivante:

```
mapred streaming -files /data/MapReduce/mapper.py,/data/MapReduce/reducer.py -mapper mapper.py -reducer reducer.py -input input -output output1
```

Afin de vérifier le bon fonctionnement de l'application, consulter l'interface web du RessourceManager http://localhost:8088 ou http://<resourcemanager_ip_address>:8088 (vérifier que le port est bien exposé dans docker-compose.yml)

Etape 4 : Test sur un dataset réel

On considère un data set décrivant des films.

Télécharger le dataset du site suivant :

https://datasets.imdbws.com/title.basics.tsv.gz

```
gunzip title.basics.tsv.gz
```

Compter le nombre de lignes

```
wc -l title.basics.tsv
```

Faire un premier test en local sur un extrait du fichier

```
tail -n 100000 title.basics.tsv > subset.tsv
```

Visualiser les 10 premières lignes

```
tail subset.tsv
```

On cherche à répondre aux questions suivantes :

1. Combien d'éléments de chaque type de film (titleType) contient le dataset?

2. Combien d'éléments par genre de film contient le dataset?

Écrire le code de map et de reduce pour répondre aux questions.

Commencer par tester en local avant de passer au cluster Hadoop.

Partie 2 :

Le but de cette partie est de découvrir la librairie python MRJob qui permet de simplifier le développement de Map et Reduce.

Présentation de mrjob

MRjob est une bibliothèque python pour MapReduce devenue actuellement très utilisée. La bibliothèque aide les développeurs à écrire du code MapReduce à l'aide du langage de programmation Python. Les développeurs peuvent tester le code MapReduce Python écrit avec mrjob localement sur leur système ou sur Hadoop.

Fonctionnement de Yield

- L'instruction yield suspend l'exécution de la fonction et renvoie une valeur à l'appelant, mais conserve suffisamment d'état pour permettre à la fonction de reprendre là où elle s'est arrêtée.
- Une fois reprise, la fonction continue son exécution immédiatement après la dernière exécution de rendement.
- Cela permet à son code de produire une série de valeurs au fil du temps, plutôt que de les calculer en une seule fois et de les renvoyer comme une liste. (Contrairement à return) !

Ex: Renvoyer plusieurs valeurs, code à tester :

```
def generatorFun():
        yield 1
        yield 2
        yield 3
# Pour tester
for value in generatorFun():
         print(value)
```

Exemple :

```
from mrjob.job import MRJob
class MRWordCount(MRJob):
        def mapper(self, _, line):
                for word in line.split():
                        yield(word, 1)
        def reducer(self, word, counts):
                        yield(word, sum(counts))
if __name__ == '__main__':
        MRWordCount.run()
```

MRStep

Le but de MRStep est d'effectuer plusieurs opérations Map et reduce dans un même programme.

Exemple:

```
from mrjob.job import MRJob
from mrjob.step import MRStep
class MRMostUsedWord(MRJob):
        def mapper_get_words(self, _, line):
                # yield each word in the line
                for word in line.strip().split():
                        yield (word.lower(), 1)
```

```
        def reducer_count_words(self, word, counts):
                # send all (num_occurrences, word) pairs to the same reducer.
                # num_occurrences is so we can easily use Python's max() function.
                yield None, (sum(counts), word)
                # discard the key; it is just None
        def reducer_find_max_word(self, _, word_count_pairs):
                # each item of word_count_pairs is (count, word),
                # so yielding one results in key=counts, value=word
                yield max(word_count_pairs)
        def steps(self):
                return [
                MRStep(mapper=self.mapper_get_words, reducer=self.reducer_count_words),
                MRStep(reducer=self.reducer_find_max_word)
                ]
if __name__ == '__main__':
        MRMostUsedWord.run()
```

Exercice1 : Expliquer le fonctionnement du code suivant.

Tester sur un dataset (exemple de fichier à créer) et vérifier que les résultats correspondent à votre compréhension du programme.

```
from mrjob.job import MRJob
class MRWordFrequencyCount(MRJob):
        def mapper(self, _, line):
                yield "chars", len(line)
                yield "words", len(line.split())
                yield "lines", 1
        def reducer(self, key, values):
                yield key, sum(values)
if __name__ == '__main__':
```

```
        MRWordFrequencyCount.run()
```

Exercice 2 : Écriture des programmes Map et Reduce avec mrjob et test en local

Vérifier que Python3 est installé sur la machine. Ensuite, il faut installer mrjob.

Si pip3 n'est pas installé sur la machine, utiliser :

```
apt install python3-pip
```

Vérifier que l'installation s'est bien passée, en tapant :

```
pip3 --version
```

Installation de mrjob

```
pip3 install mrjob
```

Dans un fichier count.py (par exemple), écrire le code mrjob comme suit :

```
from mrjob.job importMRJob
class Count(MRJob):
        def mapper(self, _, line):
                for word in line.split():
                        yield(word, 1)
        def reducer(self, word, counts):
                yield(word, sum(counts))
if__name__ =='__main__':
        Count.run()
```

Pour exécuter, tapez :

```
Python3 MapReduce/count.py file1.txt
```

Si tout va bien, on passe au cluster Hadoop.

Exercice 3 : Test sur un dataset réel

On considère un dataset décrivant des films, utilisé dans la première partie.

Extraire une partie du dataset subst.tsv comme suit.

```
tail -n 100000 title.basics.tsv > subset.tsv
```

Visualisez les 10 premières lignes

```
tail subset.tsv
```

On cherche à répondre aux questions suivantes :

1. Combien d'éléments de chaque type de film (titleType) contient le dataset?

2. Combien d'éléments par genre de film contient le dataset?

Écrire les codes du programme mrjob pour répondre aux questions.

Exercice 4 : Test de MapReduce en version python sur un cluster Hadoop

Utiliser la commande ci-dessous pour exécuter Count.py sur Hadoop qui a la forme suivante :

```
python3 <nom programme mrjob dans namenode> -r hadoop <url du fichier input> output
```

Notez qu'il a fallu utiliser l'url hdfs du fichier input dans la commande ci-dessus. Pour trouver les éléments constituant l'URL de forme hdfs://<namenode host> :port/<namenode user>/<file path>, il a fallu regarder les champs de configuration dans /etc/hadoop/core-site.xml

Par exemple, la commande serait :

```
python3 /data/MapReduce/count.py -r
                    hadoop hdfs://namenode:9000/user/root/input/file1.txt
```

Afin de vérifier le bon fonctionnement de l'application, consulter l'interface web du RessourceManager http://localhost:8088 ou http://<resourcemanager_ip_address>:8088 (vérifier que le port est bien exposé dans docker-compose.yml).