Traiter les données avec Spark

Une partie de ce document s'inspire directement de l'excellent cours de Pierre Nerzic https://perso.univ-rennes1.fr/pierre.nerzic/Hadoop/index.htm.

Implémentation pour les TP :

Spark fonctionne normalement sur un cluster, mais propose également un mode standalone pour tester les programmes qu'on va ensuite faire tourner sur un cluster. De plus, on utilise différentes API de haut niveau pour parler à Spark. Pour les TP, nous allons utiliser l'API python (https://spark.apache.org/docs/latest/api/python/index.html). Elle est apportée par le module `pyspark` de python. Quand vous installez ce module, il installe sur votre ordinateur une implémentation de Spark, avec toute l'architecture de machines virtuelles java nécessaires au fonctionnement de Spark.

Vous pourrez effectuer le TP en installant directement `pyspark` par pip :

```bash
pip install pyspark
```

Introduction

Principalement basé sur le paradigme map-reduce, Spark est un framework open source de calcul distribué, développé en 2009 par Matei Zaharia lors de son doctorat au sein de l'université de Californie à Berkeley, puis confié en 2013 à la fondation Apache. Les créateurs de Spark ont fondé en 2013-2014 la société Databricks, qui vend des services autour de Spark : plateforme web permettant de faire du calcul sur un Cloud avec une interface de type notebook.

Spark a été conçu pour répondre à quelques faiblesses de Hadoop :

- Hadoop fonctionne en mode batch : il écrit les résultats intermédiaires de ses calculs sur le disque pour permettre la communication entre mapper et reducer et augmenter la tolérance aux pannes (résilience). Mais cela coûte du temps de calcul…
- Hadoop est restreint aux opérations map et reduce ce qui limite les possibilités d'expression.

Pour répondre à ceci, Spark va chercher à garder autant que possible en mémoire vive les résultats des calculs. Mais il doit aussi veiller à la persistence des données, à priori non assurée par un stockage en RAM.

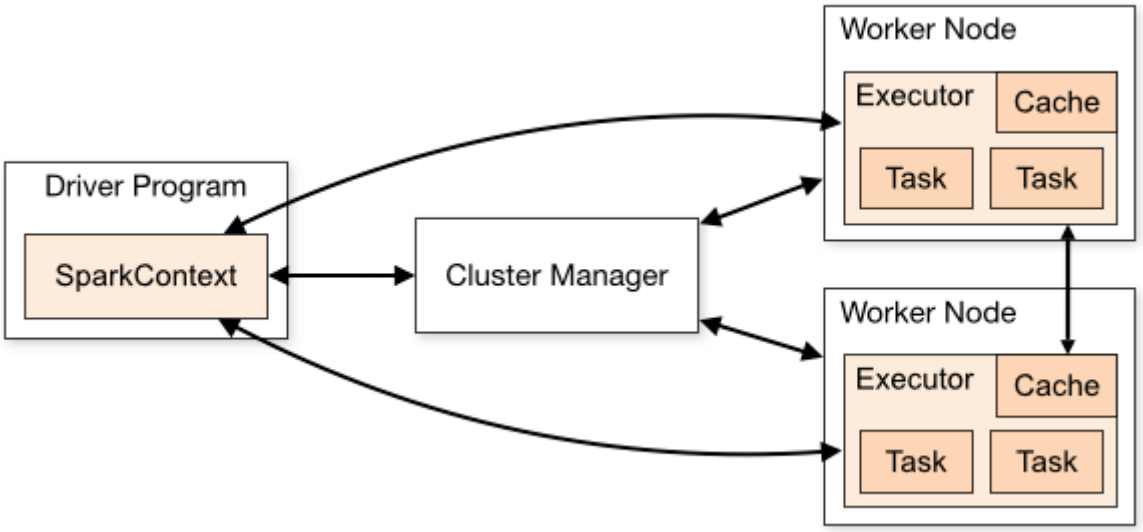
Rappel d'ordre de grandeur

	Débit	Latence
RAM	15 Go/s	10^{-8} s
SSD	0.5 Go/s	10^{-4} s

Spark propose également plusieurs bibliothèques adaptées à des domaines différents de l'analyse des données : SQL (pour le traitement de données relationnelles), GraphX (graphes) et Mlib (machine learning).

Exécution d'un programme

Un programme lance un processus `driver` sur la machine "master", qui envoie sur les différentes machines "workers" différents processus d'exécutions, eux-mêmes divisés en plusieur `tasks` . Chacune de ces tâches élémentaires est exécutés dans une machine virtuelle java.



(https://spark.apache.org/docs/latest/cluster-overview.html)

C'est le `cluster manager` qui gère la gestion des ressources entre les applications. Spark peut gérer lui-même un cluster, et dans ce cas, il dispose de son propre ordonnanceur qui distribue les différents `tasks` entre les machines du cluster. C'est le mode de fonctionnement optimal. Il nécessite des machines bien dotées en mémoire vive, mais moins nombreuses que dans un cluster Hadoop.

On peut cependant utiliser (entre autre) deux autres modes de fonctionnement :

- Spark délègue le travail à un cluster Hadoop, et utilise donc l'ordonnanceur de ce cluster.
- Comme expliqué précedemment, Spark est capable de travailler en local (option par défaut), mode "standalone" sur une seule machine. Ce mode de fonctionnement est adapté à la mise au point des algorithmes, qu'on fera ensuite tourner en grandeur nature sur un cluster.

Le passage à l'échelle (de une à plusieurs machines) est d'ailleurs entièrement gérée par le framework. C'est-à-dire qu'il n'est pas nécessaire de réécrire les programmes, l'adaptation au calcul distribué se fait tout seul. De même, que la parallélisation se fait en arrière plan sur un processeur disposant de plusieurs coeur.

Resilient Distributed Datasets : RDD

Les objets manipulés par Spark sont des **Resilient Distributed Datasets** (RDD), qui vivent dans des machines virtuelles java, et sont stockés en mémoire. Ils sont distribués entre plusieurs machines, et sont résilient, c'est-à-dire que si une machine tombe en panne, la partie du RDD qu'elle traite est reconstruite à partir de ses ancêtres.

Les opérations sur les RDD sont de deux types :

- Des transformations, qui rendent un pointeur vers un nouveau RDD ; on parle d'évaluation paresseuse (*lazy evaluation*). Parmi les transformations standard, on trouve en particulier les classiques `map`, `reduce`, `filter`.
- Des actions qui retournent une valeur au driver. C'est au moment où le processus exécute une action que les calculs sont vraiment effectués.

Construction d'un graphe pour les calculs distribués

L'ordonnanceur construit un graphe orienté acyclique (DAG : Directed Acyclic Graph) à partir des actions et transformations sur les RDD :

- Chaque noeud correspond à un RDD ou un résultat
- Chaque arête correspond à une transformation ou une action

Lorsqu'une panne survient, on peut récupérer les informations d'un noeud grâce à ses noeuds parents.

Répartition des données entre les executors

Les données sont réparties en partitions par les RDD et attribuées aux différents executors. Chaque tâche correspond au traitement d'une partition, toutes les tâches sont effectuées en parallèle. Une étape (`stage`) est terminée lorsque toutes les partitions ont été traitées (toutes les tâches sont terminées). On peut alors passer au `stage` suivant. Un enchainement d'étapes correspond à un `job` Spark, créé pour chaque action effectuée sur un RDD.

Le passage d'une étape à une autre se fait dès qu'il est question de redistribuer les données (on parle de `shuffle`). Il est donc important de repérer quelles actions entrainent ce shuffle puisqu'il va être coûteux en temps de calcul (transfert de données).

Les API (interfaces de programmation applicative)

Spark est écrit dans le langage **scala**. C'est un langage de script, fortement typé (contrairement à python), adapté à la fois à la programmation objet et à la programmation fonctionnelle (contrairement à Java). Il s'exécute dans une machine virtuelle java, et est donc bien taillé pour interagir avec java, qui est la base de Hadoop.

Spark propose plusieur API, c'est-à-dire des ensembles de classes, de méthodes ou de fonctions qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels :

- scala (API native)
- java
- python (pyspark)
- R (sparkR)

Nous travaillerons ici avec l'API `pyspark`.

On peut obtenir pyspark lorsque l'on fait une installation complète de Spark mais on peut aussi installer `pyspark` comme un module supplémentaire de la distribution anaconda. Dans ce cas, l'installeur de anaconda installe les modules python utilisés, ainsi qu'une infrastructure spark, basée sur des machines virtuelles java.

Un premier contact avec pyspark

Vous êtes renvoyés à l'introduction en tête de cette feuille, qui vous explique comment utiliser pyspark. Les epérimentations supposent que vous avez effectué l'une des opérations proposées.

Vous avez deux possibilités :

- importer le module `pyspark` dans un script python,
- utiliser la console interactive `pyspark`.

Les deux méthodes sont utiles et dépendent de l'activité que vous visez.

Import du module pyspark dans un script python

Dans le script, on doit créer un objet, soit de type `SparkContext` ou `SparkSession` par l'une ou l'autre des syntaxes :

PYTHON
```
from pyspark import SparkContext
sc = SparkContext("local", "First App")
```

ou bien

PYTHON
```
from pyspark.sql import SparkSession
spark  = SparkSession.builder.master("local").getOrCreate()
```

Ces deux objets `sc` et `spark` décrivent une connexion vers le moteur de calcul (local, ou cluster spark ou hadoop) qui effectuera les opérations qui seront demandées par la suite. Nous utiliserons ici la version `local`.

La suite des commandes fera appel à l'un ou l'autre de ces objets. La méthode `SparkSession` semble devenir la méthode standard d'accès dans les versions actuelles de spark.

La console interactive pyspark

Tapons la commande `pyspark` dans un terminal ; on obtient alors une console interactive :

SHELL
```
Python 3.7.3 (default, Mar 27 2019, 22:11:17)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
19/12/02 11:36:04 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using
builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.4.4
      /_/

Using Python version 3.7.3 (default, Mar 27 2019 22:11:17)
SparkSession available as 'spark'.
>>>
```

On peut aussi utiliser la commande `pyspark --master local[4]` qui spécifie l'utilisation de Spark en mode standalone, avec 4 processus en parallèle. De la même manière, on pourra spécifier l'adresse du cluster que l'on souhaite utiliser via l'option `--master` (voir plus loin).

La console interactive `pyspark` est simplement une console python qui chage les modules lié à spark, et crée en particulier un objet `sc` de type `SparkContext` et un objet `spark` de type `SparkSession`.

Un premier essai avec le wordcount

On part du principe que vous avez dans votre répertoire un fichier `texte`, contenant par exemple :

texte

```txt
tata toto tutu titi tata
toto tata tititi
titi
```

Exécutez la suite de commandes dans la console pyspark :

```python
lines = sc.textFile("texte")
word_count= lines.flatMap(lambda line: line.split())
word_count2 = word_count.map(lambda  word: ( word,1))
word_count3 = word_count2.reduceByKey(lambda a,b: a+b)
word_count3.first()
word_count3.take(3)
word_count3.saveAsTextFile("sortie")
```

Essayez de comprendre ce que fait chacune des commandes lancées.

On peut faire quelques remarques :

- L'objet `sc` est un objet `SparkContext` . Il est construit automatiquement au lancement du terminal interactif.
- L'objet `lines` est un RDD qui ne contient pas les données du texte, uniquement un pointeur vers celui-ci.
- Les commandes `flatMap()` , `map()` , `reduceByKey()` sont des transformations. Le calcul n'est pas effectué au moment où elles sont lancées. On transforme un RDD en un autre.
- Les commandes `first()` , `take(n)` et `collect()` sont des actions qui effectuent réellement un calcul. Elles affichents respectivement la première valeur du RDD, les `n` premières ou l'ensemble des résultats obtenus.

On note que l'on utilise ici, sur le même RDD word_count3, deux actions différentes. Chaque transformation ayant lieu avant ces actions sera donc évaluée deux fois puisque leur résultat n'est pas gardé en mémoire. A condition que son utilisation génère un réel gain de temps, la méthode `persist()` peut résoudre ce problème en permettant le stockage en cache du résultat de ces transformations.

On peut faire la même chose, mais maintenant, en utilisant un script python :

wordcount.py

```python
import sys
from pyspark import SparkContext

sc = SparkContext()
lines = sc.textFile(sys.argv[1])
word_count= lines.flatMap(lambda line: line.split())
word_count2 = word_count.map(lambda  word: ( word,1))
word_count3 = word_count2.reduceByKey(lambda a,b: a+b)
res = word_count3.collect()

for word,cpt in res:
    print(word,cpt)
```

On peut maintenant lancer ce script par

```shell
python wordcount.py texte
```

ou

```shell
spark-submit wordcount.py texte
```

SparkContext et SparkSession

Dans la version 1 de Spark, l'objet de base était les `SparkContext`. A partir de la version 2, l'accent a été mis sur un nouvel objet : les `SparkSession`.

Ces deux objets sont le point d'entrée sur un serveur (ou cluster) de calcul. Une instance de l'une ou l'autre de ces classes crée une machine virtuelle java driver qui ordonnancera la suite du calcul. Elle est également en charge de la lecture des données d'entrée.

Un objet `SparkSession` permettra de lancer tout ce que lance un `SparkContext`, mais autorise d'autres types d'objets que les `RDD` (`DataFrames`, `DataSets`), qui possède des méthodes permettant de retrouver les fonctionnalités de SQL. L'invocation d'une nouvelle `SparkSession` se fait par :

PYTHON
```
spark = SparkSession.builder.master("type_de_cluster") \
                    .appName('nom_de_la_session') \
                    .getOrCreate()
```

`master` définit le type de cluster sur lequel seront effectués les calculs. Il y a plusieurs possibilités :

- `local[n]` : lancement sur la machine locale, où `n` désigne le nombre de threads
- `yarn` : utilisation de l'ordonnanceur `yarn` de Hadoop
- `spark` : cluster Spark

`appName` définit le nom de l'application et est optionnel (Spark attribue un nom par défaut).

`getOrCreate` fabrique la session si elle n'existe pas déjà où y accède si elle existe.

Une fois la session créée, on dispose de toute les méthodes d'un `SparkContext`, auxquelles on accède par des commandes de la forme :

PYTHON
```
rdd = spark.sparkContext.function
```

où `function` représente notamment les méthodes d'accès suivantes :

SHELL
```
>>> spark.
spark.Builder            spark.newSession         spark.stop
spark.builder             spark.range              spark.streams
spark.catalog             spark.read               spark.table
spark.conf                spark.readStream         spark.udf
spark.createDataFrame    spark.sparkContext       spark.version
spark.getActiveSession    spark.sql
```

Quelques commandes pour manipuler des RDD

Les RDD sont essentiellement des paires clé/valeur, et beaucoup de fonctions reposent sur cette structure (à la différence de la structure `DataFrame`, que l'on verra plus loin). L'ensemble des commandes disponibles se trouvent dans la documentation officielle (https://spark.apache.org/docs/latest/rdd-programming-guide.html).

Opérations d'entrée-sortie

- `rdd = sc.parallelize(donnees)` insérer des données provenant de python (ou Scala)
- `rdd = sc.textFile('fichier_ou_repertoire')` rentrer un fichier texte, ou un répertoire de fichiers texte (chaque fichier est lu ligne par ligne)

- `rdd = sc.wholeTextFile('repertoire')` les fichiers sont stockés par couple : nom_fichier/contenu
- `rdd.saveAsTextFile('fichier-sortie')` écrire le contenu dans un fichier

Transformations

Les transformations ne sont pas effectuées au moment où elles sont appelées, mais juste lorsqu'une action demandera explicitement d'effectuer un calcul.

- `rdd2 = rdd1.map(fonction)` applique une fontion à un RDD : on récupère autant d'enregistrements qu'il y en avait dans le RDD de départ
- `rdd2 = rdd1.flatMap(fonction)` applique une fontion à un RDD et rend un résultat à plat
- `rdd2 = rdd1.groupByKey()` regroupe les paires clé/valeur par clés
- `rdd2 = rdd1.ReduceByKey(fonction)` regroupe les paires clé/valeur par clés, puis applique la `fonction` de réduction sur les valeurs de chaque regroupement
- `rdd2 = rdd1.sortByKey(ascending)` retourne un RDD trié, à condition d'avoir un ordre naturel sur les clés (le paramètre `ascending` vaut `True` par défaut)
- `rdd2 = rdd1.filter(fonction)` retourne les lignes du RDD qui acceptent la condition (`fonction` doit être à valeurs booléenne)
- `rdd2 = rdd1.join(rdd2)` (et de même `leftOuterJoin`, `rightOuterJoin`, `fullOuterJoin`) jointure de deux `DataSets` suivant l'égalité des clés
- `rdd2 = rdd1.distinct()` retourne le RDD sans doublons

Actions

Ces différentes actions correspondent nécessairement à une phase `reduce` et sortent un résultat calculé.

- `rdd.collect()` retourne le RDD sous forme d'une liste dans le langage de l'API utilisée (`python` si on utilise `pyspark`). Peu utilisable en pratique (dans le cas de grosses données)
- `rdd.take(n)` retourne `n` enregistrements (plus utilisable que `collect`)
- `rdd.takeOrdered(n, fonction)` retourne les n premiers éléments du RDD ordonnés selon fonction appliquée aux clés
- `rdd.first()` retourne le premier élément du RDD
- `rdd.count()` retourne le nombre d'éléments
- `rdd.reduce(fonction)` exécute une opération de `reduce` pour une fonction donnée associative (addition, max, produit, ...)

Exercice 1

Ecrire un programme spark prenant en entrée un texte, et sortant le mot le plus utilisé dans ce texte parmi les mots de plus de 5 lettres.

▾ Pour tester

Le fichier texte `hadoop_wikipedia` (https://math.univ-angers.fr/~badreau/data/hadoop_wikipedia) associé au programme `plus_frequent.py` par la commande unix :

```shell
python plus_frequent.py hadoop_wikipedia
```

doit renvoyer le résultat :

```SHELL
"hadoop"    20
```

▾ Solution

plus_frequent.py

```PYTHON
#! /usr/bin/env python3

import sys, re
from pyspark import SparkContext

motif = re.compile("\w{5,}")

sc = SparkContext()

lines = sc.textFile(sys.argv[1])
map_find_motif = lines.flatMap(lambda line: motif.findall(line))
map_word_1 = map_find_motif.map(lambda word: (word.lower(),1))
count_word = map_word_1.reduceByKey(lambda a,b: a+b)
map_count_word = count_word.map(lambda t: (t[1],t[0]))
more_frequent = map_count_word.sortByKey(False).first()

print(f"{more_frequent[1]}\t{more_frequent[0]}")
```

Exercice 2 : un exercice sur un fichier texte

Le fichier isd-history.txt (https://math.univ-angers.fr/~badreau/data/isd-history.txt) contient des données sur les bases météorologiques recensées par l'USAF. La table contient les champs suivants (les champs ne sont pas forcément tous remplis, mais les différentes données sont exactement alignées verticalement, ce qui permet de les identifier facilement) :

Nom	**description**	**colonnes**
USAF	Air Force station ID. May contain a letter in the first position	0-5
WBAN	NCDC WBAN number	7-11
STATION NAME	Designation	13-41
CTRY	FIPS country ID	43-46
ST	State for US stations	48-49
CALL	indicatif	51-55
LAT	Latitude in thousandths of decimal degrees	57-63
LON	Longitude in thousandths of decimal degrees	65-72
ELEV	Elevation in meters	74-80

Nom	description	colonnes
BEGIN	Beginning Period Of Record (YYYYMMDD)	82-89
END	Ending Period Of Record (YYYYMMDD)	91-98

Avec Spark, réaliser les opérations suivantes :

- Charger le fichier dans un RDD. Récupérer uniquement les lignes qui nous intéressent (par exemple, celles qui contiennent un chiffre en deuxième position)
- Compter le nombre d'enregistrement.
- Compter les nombre de stations dans chaque hémisphère (latitude positive ou négative)
- Déterminer les dix stations qui ont eu la période d'activité la plus grande (écart entre le début des mesures et la fin). Penser à ne regarder que les stations dont la date de fin a été renseignée.
- Déterminer le pays ayant le plus de stations
- Déterminer le nombre de pays ayant des stations

▾ Résultats attendus

TXT

```
Nombre d'enregistrement :  29705

Nombre de stations par hémisphère :
- NA    1204 stations
- Nord  24052 stations
- Sud   4449 stations

Stations ayant eu la plus longue période d'activité :
- KALAJOKI ULKOKALLA            117 ans d'activité
- PARAINEN FAGERHOLM            117 ans d'activité
- TURKU                         117 ans d'activité
- PARAINEN UTO                  117 ans d'activité
- OULU                          112 ans d'activité
- KUOPIO                        112 ans d'activité
- TAMPERE PIRKKALA              112 ans d'activité
- HANKO RUSSARO                 112 ans d'activité
- VYBORG                        112 ans d'activité
- KUUSAMO                       109 ans d'activité

Pays le plus représenté : US 7320 stations

Nombre de pays ayant des stations : 252
```

▾ Solution

PYTHON

```python
#! /usr/bin/env python3

from pyspark.sql import SparkSession

spark  = SparkSession.builder.master("local").appName('Mabadreau').getOrCreate()
input_file = "isd-history.txt"

# Récupérer les données
RDD = spark.sparkContext.textFile(input_file).filter(lambda l : len(l)>0 and l[1] in '0123456789')

# Nombre d'enregistrement
nb_stations = RDD.count()
print("\nNombre d'enregistrement : ",nb_stations)

# Nombre de stations par hémisphère
hem = RDD.map(lambda l : ("Nord",1) if l[57]=='+' else (("Sud",1) if l[57]=="-" else ("NA",1)))
nb_stat_hem = hem.reduceByKey(lambda a,b: a+b).sortByKey().collect()
print("\nNombre de stations par hémisphère :")
for hem,cpt in nb_stat_hem :
    print(f"- {hem}\t{cpt} stations")

# 10 stations qui ont eu la période d'activité la plus grande
activite = RDD.filter(lambda l : len(l)>91)
periode = activite.map(lambda l: (int(l[91:95])-int(l[82:86]),l[13:42])) # Clé = période d'activité,
Valeur = nom de la station
compte_periode = periode.sortByKey(False).take(10) # Tri décroissant sur les clés
print("\nStations ayant eu la plus longue période d'activité :")
for cpt,stat in compte_periode :
    print(f"- {stat}\t{cpt} ans d'activité")

# Pays ayant le plus de station
stations_par_pays = RDD.map(lambda l: (l[43:47],1)) # Clé = ID pays, Valeur = 1
nb_par_pays = stations_par_pays.reduceByKey(lambda a,b:a+b).map(lambda t:(t[1],t[0])) # Clé = Nombre de
station, Valeur = ID pays
pays_plus_representes = nb_par_pays.sortByKey(False).first() # Tri décroissant sur les clés
print(f"\nPays le plus représenté : {pays_plus_representes[1]}\t{pays_plus_representes[0]} stations")

# Nombre de pays possédant des stations
pays = RDD.map(lambda l: l[43:47]).distinct().filter(lambda p: p.strip()!="")
nb_pays = pays.count()
print(f"\nNombre de pays ayant des stations : {nb_pays}")
```

RDD, DataFrame et Dataset

L'objet de base de Spark était initialement le RDD (resilient distributed dataset), collection d'objets non structurés, tout à fait dans la logique NoSQL.

Dans un second temps, les concepteurs de Spark ont introduit les DataFrames, pour retrouver un peu la logique des tables SQL. Un DataFrame est un RDD dont les objets sont de type `row`.

Enfin, ils ont voulu introduire un objet plus structuré, avec des colonnes typées. Ce sont les DataSets (nom un peu embrouillant, car induisant une confusion avec les RDD). Malheureusement, du fait que python est typé dynamiquement, cette structure n'est actuellement pas implémentée dans l'API python `pyspark` ou dans l'API R. Pour les exploiter, il faudrait utiliser des API dans des langages typés statiquement, comme java ou scala, ce que nous ne ferons pas dans ce cours.

La structure de DataFrame

Un DataFrame est une structure voisine fonctionnellement des DataFrames de pandas ou de R, mais qui est adapté à la parallélisation.

Un DataFrame a un schéma, soit défini explicitement, soit découlant de la structure des données (par exemple dans le cas d'une donnée au format `json`).

Un exemple de manipulation élémémentaire

Ici, on travaille à partir du fichier agents.json (https://math.univ-angers.fr/~badreau/data/agents.json).

Regardez la structure du fichier en question, puis exécutez les commandes une à une pour comprendre ce qu'elles font.

PYTHON

```python
from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local[4]").getOrCreate()

df = spark.read.load("agents.json",format="json")
df.show()
df.printSchema()
df.count()

df2 = df.filter(df['country_name'] == 'India').filter(df['sex'] == 'Female')
df2.count()

df.groupBy('country_name').count().show()

df.select("id","country_name").show()

df.createOrReplaceTempView("people")
sq = spark.sql("SELECT * FROM people")
sq.show()
```

Chargement depuis un RDD

On peut fabriquer un `DataFrame` à partir d'un RDD (et inversement). Récupérons le fichier eleves.csv (https://math.univ-angers.fr/~badreau/data/eleves.csv) puis effectuons les opérations ci-dessous :

PYTHON

```python
from pyspark.sql import SparkSession, Row

spark = SparkSession.builder.master("local").getOrCreate()
sc = spark.sparkContext

lines = sc.textFile("eleves.csv")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[1], date_naissance=int(p[4])))

schemaPeople = spark.createDataFrame(people)
schemaPeople.createOrReplaceTempView("people")

nes_en_1995 = spark.sql("SELECT name FROM people WHERE date_naissance = 1995")

nes_en_1995.collect()
nes_en_1995.show()
```

Utilisation de l'API SparkSQL

`SparkSQL` permet d'accéder à des données de plusieurs `DataSets` en utilisant une logique fonctionnelle similaire à SQL.

Soit une table de clients `(idclient, nom)` et une table d'achats `(idachat, idclient, montant)`. On veut afficher les noms des clients ayant fait au moins un achat d'un montant supérieur à 30€.

En SQL, on ferait une requête :

SQL

```sql
SELECT DISTINCT nom FROM achats JOIN clients
ON achats.idclient = clients.idclient
AND achats.montant > 30.0;
```

En `pySparkSQL`, si on dispose de deux dataframes `achats` et `clients`, on construit un nouveau DataFrame par :

PYTHON

```
resultat = achats.filter(achats.montant > 30.0)\
.join(clients, clients.idclient == achats.idclient)\
.select("nom")\
.distinct()
```

On dispose d'une méthode `groupBy`. On applique ensuite une fonction d'agrégation type `count`, `sum`, `avg`, `min`, ou encore `max`.

PYTHON

```
tapc = achats.groupBy("idclient").sum("montant")
napc = achats.groupBy("idclient").count()
```

Ces fonctions d'agrégation vont créer une colonne nommée `"SUM(montant)"` et `"COUNT(montant)"`.

Pour trouver le client qui a acheté le plus :

PYTHON

```
topa = achats.groupBy("idclient").sum("montant") \
        .sort(desc("SUM(montant)")).first()
```

Exercice 3

Refaire l'exercice précédent (exercice 2 sur les stations météo) en travaillant désormais à partir d'un dataframe.

- [FACULTATIF] Transformer le fichier isd-history.txt (https://math.univ-angers.fr/~badreau/data/isd-history.txt) à partir d'un programme python pour obtenir le fichier `csv` (https://math.univ-angers.fr/~badreau/data/isd-history.csv) correspondant aux observations sur les stations météos :

USAF	WBAN	STATION NAME	CTRY	ST	CALL	LAT	LON	ELEV(M)	BEGIN	END
007018	99999	WXPOD 7018				0	0	7018	20110309	20130730
007026	99999	WXPOD 7026	AF			0	0	7026	20120713	20170822

- Charger les données dans un nouveau script python en utilisant le module `pyspark`, ou directement dans la console interactive, par la commande :

PYTHON

```
df = spark.read.load("isd-history.csv",format="csv", sep=",", inferSchema="true", header="true")
```

- Reprendre les questions de l'exercice 2

Machine Learning avec Spark et `pyspark`

Spark propose deux modules de Machine Learning : `Mlib` et `ML`. Nous nous focaliserons sur `ML` car `Mlib` n'est plus activement développé, et doit être à terme remplacé par `ML`. Cette bibliothèque implémente en gros les fonctionnalités du module python `scikit-learn`, mais de façon adaptée à un traitement distribué.

On vous propose un TP extrait du site companion (https://github.com/drabastomek/learningPySpark) du livre de Tomasz Drabas et Denny Lee Learning PySpark (https://www.amazon.fr/Learning-PySpark-Tomasz-Drabas/dp/1786463709). Il s'agit de

prévoir les chances de survie d'un enfant en fonction des conditions qui ont entouré sa naissance.

On dispose d'un fichier births_transformed.csv.gz (https://math.univ-angers.fr/~badreau/data/births_transformed.csv.gz) recensant un grand nombre de naissances. Téléchargez le, et regardez comment il est constitué.

Ensuite, lancez le notebook (https://math.univ-angers.fr/~badreau/data/ML.ipynb).

Index (https://math.univ-angers.fr/~badreau/)

Last updated 2023-10-11 13:12:36 +0200