Practical Guide of PySpark for Data Engineer: Common Functions and Application Examples

Keita, Moussa

June 2022

Guide Pratique de PySpark pour Data Engineer

Fonctions Usuelles et Exemples d'Applications

Moussa KEITA

Expert Data/Formateur
Big Data–Data Science

Consultant Data à EDF, Société Générale, Caisse des Dépôts

Paris

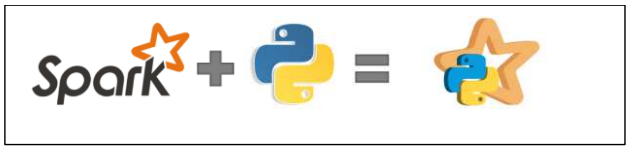
Contact Email : keitam09@ymail.com

Document version 1.0
(Juin 2022)

SOMMAIRE

1 INTRODUCTION

Apache Spark est un moteur de traitement en mémoire permettant d'appliquer des calculs distribués sur de grosses volumétries de données stockées sur un cluster. Spark est framework applicatif disponible dans plusieurs langages de programmation sous forme d'API : Scala (SparkScala), Java (JavaSpark), Python (PySpark), R (SparkR), etc. Dans ce document, nous présentons le cas particulier de PySpark qui est l'implémentation du framework en langage python.



Il est à noter que le langage Scala reste le langage d'écriture natif d'un traitement Spark Scala. Mais l'API PySpark s'est très rapidement imposé en raison de sa très grande popularité dans la communauté des Data Engineers. Cependant il faut rappeler que quel que soit l'API utilisé pour implémenter un traitement Spark (Scala, R, Python, Java), l'exécution du traitement se fait toujours à l'intérieur d'un processus JVM (Java Virtual Machine). Cela signifie que pour que le code d'un traitement Spark puisse s'exécuter, celui-ci doit d'abord être compilé en bytecode Java et soumis à exécution dans le JVM. Par exemple, lorsque le traitement Spark est codé en langage Scala, c'est le compilateur scalac qui convertit le code source en bytecode Java. De même lorsque le traitement Spark est codé en Java, c'est le Java Compiler qui se charge de convertir le code source en. Mais pour le cas de l'API PySpark, c'est par l'intermédiaire du package Py4J, que les traitements codés en Python s'exécutent sur la JVM. Py4J est une librairie Java intégrée à PySpark et permet à Python d'interagir dynamiquement avec des objets JVM. C'est grâce à cette fonctionnalité d'interfaçage que l'interpréteur Python peut appeler un objet Java spécifique (ex : classe, méthode, collection) pour chaque objet Python traité. Ainsi, malgré la diversité des APIs permettant d'implémenter des traitements Spark, l'exécution de ces derniers nécessite la présence d'un JVM. Par conséquent, l'exécution de tout traitement Spark sur un environnement donné nécessite que le package Java soit préalablement installé. En effet, Java reste le langage primitif d'exécution de Spark. Cependant, grâce à la multiplicité des APIs, les Data Engineers ont la possibilité d'implémenter leurs traitements Spark dans le langage de leur choix sans avoir à se préoccuper des mécanismes qui permettent de convertir leur code en bytecode exécutable dans le JVM.

L'objectif du présent document est de présenter les fonctions de traitement Spark couramment rencontrées lors d'une opération de Data Engineering. Chaque fonction présentée est illustrée par un exemple d'application concret. Tous les exemples présentés sont codés en utilisant la version 3.6 de Python et sont testés sur une plateforme CDH version 7.1 sur laquelle est installée Spark version 2.4 et Hadoop version 3.1.

2 CREER UNE SESSION SPARK

Le codage et l'exécution de tout traitement Spark commence par l'initialisation d'une SparkSession. La session Spark est le point d'entrée qui offre toutes les fonctionnalités pour interagir avec les objets Spark notamment l'accès aux fonctions de traitement de données ainsi que la configuration de l'environnement nécessaire à l'exécution de ces fonctions. L'exemple ci-dessous illustre l'initialisation d'une session Spark.

2.1.1 Initialiser une session Spark :la fonction SparkSession.builder

La fonction SparkSession.builder permet d'initialiser une session Spark avec la méthode getOrCreate(). Le code ci-dessous montre comment créer une session Spark Standard avec la configuration minimale.

```
#Initialisation de la session Spark
import pyspark
from pyspark.sql import SparkSession

#Spécification minimale
spark =
SparkSession.builder.appName("mysparkjob").enableHiveSupport().getOrCreate(
)
#Spécification du master
spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
```

L'appel de la méthode SparkSession.builder s'accompagne d'autres méthodes dont certaines sont obligatoires et d'autres optionnelles.

La méthode *appName()* permet d'attribuer un nom à l'application Spark.

La méthode enableHiveSupport() est une fonction optionnelle qui active les fonctionnalités permettant d'interagir avec les objets Hive à l'intérieur d'une application Spark. L'appel de cette option permet de créer une connection entre Spark et le métastore Hive de sorte que tous les objets Hive (databases et tables) sont accessibles depuis la session Spark.

La méthode *getOrCreate()* permet de créer, à proprement parler, l'objet SparkSession. Il s'agit d'une fonction d'initialisation de la sessions Spark. Et lorsque l'objet Spark existe déjà, c'est cette instance qui est renvoyée.

La méthode *master()* permet d'indiquer l'adresse du nœud master du cluster Spark. Ici nous spécifions "yarn", car nous utilisons YARN comme cluster Manager. Mais dans d'autres situations comme par exemple le cas d'un cluster standalone, il est possible de spécifier l'adresse physique du nœud master. Ex : *master("spark://spark:7077")*.

Aussi lorsque, le nœud master se situe sur la machine où le job Spark est exécuté, on peut utiliser la valeur *master("local[*]")*.

NB : Il n'est pas obligatoire de spécifier le *.master()* lors de l'initialisation de la session Spark. Le master peut être défini dans les confs lors de l'exécution du job Spark à travers le spark-submit. Nous reviendrons plus tard sur le spark-submit.

2.1.2 Ajouter des configurations à la session Spark : la fonction conf.set().

Lorsque la session Spark est initialisée, elle est configurée, par défaut, soient avec les configurations fournies par les fichiers de confs du cluster Spark, soient avec les configurations spécifiées en *--conf* lors de l'exécution du job Spark en spark-submit. (Nous reviendrons plus tard sur l'exécution d'un job Spark dans le chapitre « Lancer le job PySpark »). Mais il est possible de spécifier explicitement les configurations de la session Spark en utilisant la méthode conf.set() sur l'objet SparkSession crée avec SparkSession.builder. L'exemple ci-dessous montre l'ajout de quelques configurations à la session Spark.

```
#Initialisation de la session Spark
import pyspark
from pyspark.sql import SparkSession
spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
# Exemples d'ajout de quelques conf à la sessions Spark
spark.conf.set('spark.sql.sources.partitionOverwriteMode', 'dynamic')
spark.conf.set("spark.es.net.ssl", "true")
spark.conf.set("spark.datasource.hive.warehouse.read.mode",
"DIRECT_READER_V2")
```

La Liste des configurations utilisables dans une session Spark sont disponibles à ce lien https://spark.apache.org/docs/2.4.0/configuration.html

NB : Lorsqu'une conf est ajoutée à l'objet SparkSession telle qu'indiquée dans les exemples ci-dessus, la valeur de cette configuration devient prioritaire par rapport à toutes les autres valeurs par défaut issues des fichiers de confs Spark ou les confs fournies par le spark-submit lors du lancement du job Spark. C'est pourquoi, le choix des confs à setter lors de l'initialisation de la session Spark doit être fait en connaissance de cause.

3 PYSPARK RDD

3.1 Créer un RDD : les fonctions parallelize(), textFile() et wholeTextFiles()

Les RDD sont généralement crées de deux manières: soit paralléliser une collection de données existante, soit charger un jeu de données disponible dans un système de stockage externe (HDFS, S3, etc.) sous un format bien défini (fichiers de texte, csv, base de données). Les exemples ci-dessous montrent trois approches de création d'un RDD.

3.1.1 Créer un RDD à partir d'une collection de données : la fonction parallelize()

L'exemple ci-dessous illustre l'utilisation de la fonction parallelize() pour créer un RDD.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [ (1,2,3.5,"2005-04-02 23:53:47"),
        (1,29,3.5,"2005-04-02 23:31:16"),
        (1,32,3.5,"2005-04-02 23:33:39"),
        (2,3926,4.0,"2000-11-21 15:34:49"),
        (2,3927,5.0,"2000-11-21 15:32:28"),
        (2,3928,5.0,"2000-11-21 15:32:56"),
        (3,32,4.0,"1999-12-11 13:14:07"),
        (3,50,5.0,"1999-12-11 13:13:38"),
        (3,160,3.0,"1999-12-14 12:54:08")
                                        ]

#Create RDD from parallelize
rdd=spark.sparkContext.parallelize(data)
# Afficher le nombre d'éléments du RDD
print(str(rdd.count()))
# Boucle pour afficher chaque élément du RDD
rddElements=rdd.collect()
for e in rddElements:
    print(e)
```

```
# output print count
9
# Output print : éléments du RDD
(1, 2, 3.5, '2005-04-02 23:53:47')
(1, 29, 3.5, '2005-04-02 23:31:16')
(1, 32, 3.5, '2005-04-02 23:33:39')
(2, 3926, 4.0, '2000-11-21 15:34:49')
(2, 3927, 5.0, '2000-11-21 15:32:28')
(2, 3928, 5.0, '2000-11-21 15:32:56')
(3, 32, 4.0, '1999-12-11 13:14:07')
(3, 50, 5.0, '1999-12-11 13:13:38')
(3, 160, 3.0, '1999-12-14 12:54:08')
```

3.1.2 Créer un RDD à partir d'un fichier de texte : la fonction textFile() :

Lorsque nous disposons d'un fichier de texte plat, nous pouvons créer le RDD en utilisant la méthode textFile() associée à la méthode sparkContext() de l'objet SparkSession.

L'exemple ci-dessous illustre l'utilisation de la fonction textFile()

Fichier de text nommé movie_desc1.txt stocké sur hdfs

```
1,"Toy Story (1995)","Adventure|Animation|Children|Comedy|Fantasy"
2,"Jumanji (1995)","Adventure|Children|Fantasy"
3,"Grumpier Old Men (1995)","Comedy|Romance"
4,"Waiting to Exhale (1995)","Comedy|Drama|Romance"
5,"Father of the Bride Part II (1995)","Comedy"
6,"Heat (1995)","Action|Crime|Thriller"
7,"Sabrina (1995)","Comedy|Romance"
8,"Tom and Huck (1995)","Adventure|Children"
9,"Sudden Death (1995)","Action"
10,"GoldenEye (1995)","Action|Adventure|Thriller"
11,"American President, The (1995)","Comedy|Drama|Romance"
12,"Dracula: Dead and Loving It (1995)","Comedy|Horror"
```

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
```

```
# Data :
#Create RDD from external Data source
rdd =
spark.sparkContext.textFile("hdfs://nnde01/user/mk/text data1/movie desc1.t
xt")
rddElements=rdd.collect()
for e in rddElements:
    print(e)
```

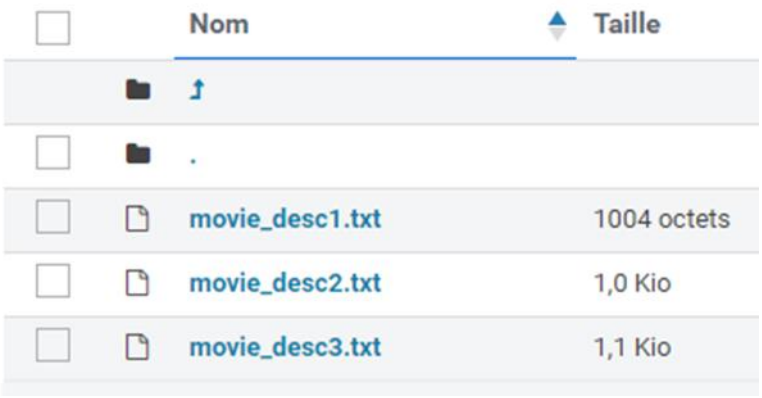
```
## Output
1,"Toy Story (1995)","Adventure|Animation|Children|Comedy|Fantasy"
2,"Jumanji (1995)","Adventure|Children|Fantasy"
3,"Grumpier Old Men (1995)","Comedy|Romance"
4,"Waiting to Exhale (1995)","Comedy|Drama|Romance"
5,"Father of the Bride Part II (1995)","Comedy"
6,"Heat (1995)","Action|Crime|Thriller"
7,"Sabrina (1995)","Comedy|Romance"
8,"Tom and Huck (1995)","Adventure|Children"
9,"Sudden Death (1995)","Action"
10,"GoldenEye (1995)","Action|Adventure|Thriller"
11,"American President, The (1995)","Comedy|Drama|Romance"
12,"Dracula: Dead and Loving It (1995)","Comedy|Horror"
13,"Balto (1995)","Adventure|Animation|Children"
14,"Nixon (1995)","Drama"
15,"Cutthroat Island (1995)","Action|Adventure|Romance"
16,"Casino (1995)","Crime|Drama"
17,"Sense and Sensibility (1995)","Drama|Romance"
18,"Four Rooms (1995)","Comedy"
19,"Ace Ventura: When Nature Calls (1995)","Comedy"
20,"Money Train (1995)","Action|Comedy|Crime|Drama|Thriller"
21,"Get Shorty (1995)","Comedy|Crime|Thriller"
```

3.1.3 Créer un RDD à partir de plusieurs fichiers textes : la fonction wholeTextFiles()

La fonction wholeTextFiles() permet de générer un RDD à partir d'un répertoire hdfs contenant un ensemble de fichiers textes. Le RDD renvoyé est un PairRDD dont les clés sont les chemins de chaque fichier texte et la valeur le contenu complet de chaque fichier.

L'exemple suivant illustre l'utilisation de la fonction wholeTextFiles() pour créer un RDD.

Trois fichiers txt stockés sur hdfs



	Nom	Taille
	↑	
	.	
	movie_desc1.txt	1004 octets
	movie_desc2.txt	1,0 Kio
	movie_desc3.txt	1,1 Kio

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

rdd = spark.sparkContext.wholeTextFiles("hdfs://nnde01/user/mk/text_data1")
rddElements=rdd.collect()
for e in rddElements:
    print("element key :"+e[0]+"\n"+ "Element value :"+e[1])
```

```
## Output
Element key : hdfs://nnde01/user/mk/text_data/movie_desc1.txt
Element value :
1,"Toy Story (1995)","Adventure|Animation|Children|Comedy|Fantasy"
2,"Jumanji (1995)","Adventure|Children|Fantasy"
3,"Grumpier Old Men (1995)","Comedy|Romance"
...
...
...
Element key : hdfs://nnde01/user/mk/text_data/movie_desc2.txt
Element value :
22,"Copycat (1995)","Crime|Drama|Horror|Mystery|Thriller"
23,"Assassins (1995)","Action|Crime|Thriller"
24,"Powder (1995)","Drama|Sci-Fi"
...
...
...
Element key : hdfs://nnde01/user/mk/text_data/movie_desc3.txt
Element value :
43,"Restoration (1995)","Drama"
44,"Mortal Kombat (1995)","Action|Adventure|Fantasy"
45,"To Die For (1995)","Comedy|Drama|Thriller"
...
...
...
```

3.1.4 Créer un RDD vide : la fonction emptyRDD()

L'exemple suivant illustre l'utilisation de la fonction emptyRDD() pour créer un RDD vide.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

# méthode 1
emptyRdd = spark.sparkContext.emptyRDD()
# Méthode 2
emptyRdd= spark.sparkContext.parallelize([])
```

3.2 Transformations d'un RDD

3.2.1 Appliquer un traitement sur chaque élément d'un RDD : les fonction map() et flatMap()

La transformation map() est utilisée pour appliquer des opérations de traitement sur chaque élément du RDD afin de renvoyer un nouvel élément. La fonction map() transforme le RDD de sorte qu'à chaque élément d'entrée correspond un et un seul élément de sortie. Quant à la flatMap(), elle permet de transformer le RDD de sorte à renvoyer pour chaque élément en entrée du RDD un ou plusieurs éléments en sortie. Les fonction map() et flatMap() sont deux fonctions utilisées pour appliquer la plupart des opérations de transformations sur un RDD : ajout ou modification de la valeur des éléments individuels du RDD.

Les exemples ci-dessous illustrent l'utilisation des fonctions map() et flatMap().

3.2.1.1 Cas du map()

Supposons un fichier de données nommé employees1.txt contenant sur chaque ligne les informations sur les employés comme suit : id_employe, genre, age, salaire et prime

```
0001  Male  19  15  39
0002  Male  21  15  81
0003  Female  20  16  6
0004  Female  23  16  77
0005  Female  31  17  40
0006  Female  22  17  76
0007  Female  35  18  6
0008  Female  23  18  94
0009  Male  64  19  3
0010  Female  30  19  72
0011  Male  67  19  14
0012  Female  35  19  99
```

On souhaite créer un RDD à partir de ce fichier et multiplier le salaire par 100 et la prime par 10. Cette transformation peut être utilisée en utilisant une fonction map(). Voir l'exemple ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

rdd0 =
spark.sparkContext.textFile("hdfs://nnde01/user/mk/text_data2/employees1.tx
t")

rdd1=rdd0.map(lambda x: x.split("  "))
rdd2=rdd1.map(lambda x: [x[0], x[1] , float(x[3])*100 , float(x[4])*10] )

rddElements=rdd2.collect()
for e in rddElements:
    print(e)
```

```
# Output
['0001', 'Male', 1500.0, 390.0]
['0002', 'Male', 1500.0, 810.0]
['0003', 'Female', 1600.0, 60.0]
['0004', 'Female', 1600.0, 770.0]
['0005', 'Female', 1700.0, 400.0]
...
...
...
```

Dans cet exemple, nous utilisons dans un premier temps la fonction map() pour splitter chaque ligne du rddo suivant le séparateur (" "). Le split transforme chaque ligne en une liste d'éléments. Ensuite, nous utilisons une deuxième fois la fonction map() pour effectuer les calculs permettant d'aboutir au résultat souhaité (multiplier le salaire par 100 et la prime par 10). Pour cela, on concatène le premier et le deuxième élément de la liste. Et nous multiplions le salaire initial (quatrième élément) par 100 et la prime par 10.

3.2.1.2 Cas du flatMap()

Contrairement à la fonction map() qui renvoie un élément en sortie pour chaque élément en entrée, la fonction flatMap() renvoie un ou plusieurs éléments en sortie pour chaque élément en sortie.

L'exemple ci-dessous illustre un cas d'utilisation de la fonction flatMap().

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
data=[("Lion,Eléphant,Tigre,Zèbre"),
        ("Crocodile,Serpent,Aigle"),
        ("Dauphin,Baleine"),
        ("Abeille,Fourmis "),
        ("Grenouille")
        ]

rdd=spark.sparkContext.parallelize(data)
rdd1=rdd.flatMap(lambda x: x.split(","))

rddElements=rdd1.collect()
for e in rddElements:
    print(e)
```

```
# Output
"Lion"
"Eléphant"
"Tigre"
"Zèbre"
"Crocodile"
"Serpent"
"Aigle"
"Dauphin"
```

```
"Baleine"
"Abeille"
"Fourmis"
"Grenouille"
```

Dans cet exemple, la fonction lambda split() de python découpe chaque ligne du rdd initial en utilisant comme séparateur ",". Ensuite la fonction flatMap() de Spark créer une ligne de RDD pour chaque mot de la ligne initiale découpée. En somme la fonction flatMap renvoie pour 1 ligne du RDD initial 1 ou n lignes dans le RDD de sortie.

3.2.2 Construire un pair RDD

Un pair RDD est un RDD dont les éléments sont de type tuple (clé, valeur). La valeur associée à chaque clé peut être de n'importe quel type : string, int, decimal, liste de valeurs, dictionnaire, etc. Un pair RDD peut être construit de plusieurs manières : soit appliquer la fonction parallelize() sur une séquence de tuples (clé, valeur), soit appliquer une fonction map() ou flatMap() sur un RDD existant et renvoyer un nouveau RDD dont les éléments sont des tuples (clé, valeurs). Les exemples ci-dessous illustrent la création d'un pair RDD.

3.2.2.1 Construire un pair RDD à partir d'une collection de tuples (clé, valeur)

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('X1', 200), ('X2', 500), ('X3', 250), ('X4', 150), ('X5', 420)]

pairRDD = spark.sparkContext.parallelize(data)
rddElements = pairRDD.collect()
for e in rddElements:
    print(e)
```

```
# Output
('X1', 200)
('X2', 500)
('X3', 250)
('X4', 150)
('X5', 420)
```

Les clés du pairRDD sont formés à partir des chaînes de caractères "X1", "X2", "X3", "X4" et "X5".

3.2.2.2 Construire un pair RDD à partir d'un RDD plat

Nous disposons d'un RDD dont les éléments sont une liste de valeurs numériques. Nous souhaitons créer un Pair RDD dont les clés correspondent aux reste de la division par 3 des valeurs numérique modulo(3). Les étapes pour construire ce pair RDD sont détaillées ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [ 200,436,136,791,70,190,561,272,174]
rdd0=spark.sparkContext.parallelize(data)
pairRDD=rdd0.map(lambda x: (x%3,x) )
rddElements = pairRDD.collect()
for e in rddElements:
    print(e)
```

```
# Output
(2, 200)
(1, 436)
(1, 136)
(2, 791)
(1, 70)
(1, 190)
(0, 561)
(2, 272)
(0, 174)
```

3.2.3 Construire un RDD plat à partir d'un pair RDD

Pour obtenir un RDD plat à partir d'un pair RDD, il suffit d'utiliser une fonction map() ou flatMap() sur le pair RDD pour appliquer la règle de transformation permettant d'aboutir aux éléments du RDD de sortie. L'exemple ci-dessous permet d'obtenir un RDD plat à partir d'un pair RDD.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('X1', 200), ('X2', 500), ('X3', 250), ('X4', 150), ('X5', 420)]
pairRDD = spark.sparkContext.parallelize(data)

# Construire RDD plat : concatener la clé et la valeur
rdd1 = pairRDD.map(lambda x: x[0].lower() + "_" + str(x[1]))

rddElements = rdd1.collect()
for e in rddElements:
    print(e)
```

```
# Output
"x1_200"
"x2_500"
"x3_250"
"x4_150"
"x5_420"
```

3.2.4 Agréger les éléments d'un RDD : les fonctions reduce() et reduceByKey()

La fonction reduce() permet d'agréger les éléments d'un RDD en appliquant une fonction cumulative : somme, produit, puissance, etc. Quant à la fonction reduceByKey(), elle permet d'appliquer une fonction cumulative sur un PairRDD. Les exemples ci-dessous illustrent l'utilisation des fonctions reduce() et reduceByKey().

3.2.4.1 Cas de la fonction reduce()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
```

```
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()


data = [ 2,4,6,1,7,10,5,2,1]
rdd0=spark.sparkContext.parallelize(data)

v=rdd0.reduce(lambda a,b: a+b) # somme cumulative
print(v)
```

```
# Output
28
```

Dans cet exemple, nous calculons la somme des éléments du RDD.

3.2.4.2 Cas de la fonction reduceByKey()

La fonction reduceByKey() applique de façon cumulative la fonction spécifiée en paramètre. Dans l'exemple ci-dessous, nous calculons simplement la somme des valeurs associées à une clé unique. Cela est matérialisé par l'opération *lambda a,b: a+b*. Bien entendu, il est possible de spécifier une autre fonction selon le cas d'usage. Par exemple *lambda a,b: a*b* ou même *lambda a,b: a^b*.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
#data
data = [('Livre', 1),('Cahier', 1),('Crayon', 1),('Gomme', 1),
('Stylo', 1),('Gomme', 1),('Livre', 1),('Cahier', 1), ('Crayon', 1),
('Stylo', 1),('Gomme', 1), ('Livre', 1),('Cahier', 1)]
#Create rdd
rdd0=spark.sparkContext.parallelize(data)
rdd1=rdd0.reduceByKey(lambda a,b: a+b)

rddElements = rdd1.collect()
for e in rddElements:
    print(e)
```

```
# Output
('Crayon', 2)
('Cahier', 3)
('Gomme', 3)
('Livre', 3)
('Stylo', 2)
```

3.2.5 Grouper les éléments suivant les clés d'un pair RDD : la fonction groupBykey()

La fonction groupByKey() permet de grouper les éléments d'un pair RDD suivant les valeurs des clés. Le groupage des données est fait de sorte que tous les éléments ayant la même valeur de clé forment un groupe. Par exemple, si l'on a un pair RDD formé par trois éléments tels que (k1,v1), (k1,v2) et (k2,v3). Après avoir appliqué la fonction groupByKey(), les éléments du pair RDD deviennent (k1,[v1,v2]) et (k2,[v3]).

Le groupage des données grâce à la fonction groupByKey() permet par la suite d'appliquer des fonctions d'agrégation sur les éléments d'un même groupe.

L'exemple ci-dessous illustre l'utilisation de la fonction groupByKey() sur un pair RDD.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
#data
#data
data = [('Livre', 1),('Cahier', 2),('Crayon', 1),('Gomme', 1),
            ('Stylo', 1),('Gomme', 3),('Livre', 1),('Cahier', 1),
('Crayon', 2),
            ('Stylo', 2),('Gomme', 1), ('Livre', 2),('Cahier', 1)]
#Create rdd
rdd0=spark.sparkContext.parallelize(data)
rdd1=rdd0.groupByKey() # Rdd groupé suivant les clés

# Affichage les valeur
print(rdd1.mapValues(list).collect()) # renvoie [('Livre',[
1,1,3]),('Cahier', [2, 1,1]),('Crayon', [1,2]),('Gomme',[ 1,3,1]),
('Stylo', [1,2])]
# Afficher le nombre d'éléments par clé
print(rdd1.mapValues(len).collect()) # renvoie
[('Livre',3),('Cahier',3),('Crayon', 2),('Gomme',3),
('Stylo', 3)]

rdd2=rdd1.groupByKey().reduceByKey(lambda a,b: a+b) # Calcul la somme des
éléments dans chaque groupe et renvoie un nouveau pair RDD
```

Après avoir appliqué la fonction groupByKey() sur un pair RDD, on peut par la suite appliquer mapValues() afin de pouvoir extraire une information agrégée à partir des

éléments par groupe. Par exemples : renvoyer la liste des éléments, renvoyer le nombre d'éléments, le max, le min, etc… Mais d'une manière générale pour obtenir les informations agrégées sur les éléments d'un groupe obtenu par groupByKey(), on utilise les fonctions d'agrégation reduceByKey(). Voir l'exemple d'utilisation de la fonction reduceByKey précédemment présenté.

3.2.6 Appliquer un filtre sur le RDD : la fonction filter()

La transformation filter() est utilisée pour filtrer les éléments d'un RDD répondant à une certaine condition. Par exemple, nous disposons d'un fichier de données qui se présente comme suit :

```
0001  Male  19  15  39
0002  Male  21  15  81
0003  Female  20  16  6
0004  Female  23  16  77
0005  Female  31  17  40
0006  Female  22  17  76
0007  Female  35  18  6
0008  Female  23  18  94
0009  Male  64  19  3
0010  Female  30  19  72
0011  Male  67  19  14
0012  Female  35  19  99
```

Ces informations correspondant respectivement à l'id_employe, le genre, l'âge, le salaire et la prime. Nous souhaitons créer un RDD à partir des lignes de texte contenant uniquement le mot « Male ». Pour aboutir à ce résultat nous utilisons la fonction filter().

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
rdd0 =
spark.sparkContext.textFile("hdfs://nnde01/user/mk/text_data2/employees1.tx
t")

#Application du filtre
rdd1=rdd0.filter(lambda x: 'Male' in x )

rddElements=rdd1.collect()
for e in rddElements:
    print(e)
```

```
# Output
0001  Male  19  15  39
0002  Male  21  15  81
0009  Male  64  19  3
0011  Male  67  19  14
0015  Male  37  20  13
0016  Male  22  20  79
0018  Male  20  21  66
0019  Male  52  23  29
0021  Male  35  24  35
...
...
...
```

3.2.7 Trier les éléments d'un RDD : les fonctions sortBy() et sortByKey()

La fonction sortBy() permet de trier les éléments d'un RDD. Et lorsque le RDD est de type Pair RDD, c'est-à-dire un RDD dont les éléments se présentent sous la forme d'un couple (clé, valeur), on peut utiliser la fonction sortByKey(). Les exemples ci-dessous montrent l'utilisation des fonction sortBy() et sortByKey() sur un RDD et un pair RDD.

3.2.7.1 Cas du sortBy()

Soit un fichier de données présenté comme suit :

```
0001  Male  19  15  39
0002  Male  21  15  81
0003  Female  20  16  6
0004  Female  23  16  77
0005  Female  31  17  40
0006  Female  22  17  76
0007  Female  35  18  6
0008  Female  23  18  94
0009  Male  64  19  3
0010  Female  30  19  72
0011  Male  67  19  14
0012  Female  35  19  99
```

Ces informations correspondant respectivement à l'id_employe, le genre, l'âge, le salaire et la prime. Nous allons trier ce RDD selon la valeur croissante de l'âge (position 3). L'exemple ci-dessous montre comment cette opération peut être effectuée.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json
```

```
spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
rdd0 =
spark.sparkContext.textFile("hdfs://nnde01/user/mk/text_data2/employees1.tx
t")
rdd1=rdd0.sortBy(lambda x: x.split("  ")[2],ascending=True) # l'âge se
trouve à la position 3 (index 2) dans la liste

rddElements=rdd1.collect()
for e in rddElements:
    print(e)
```

```
# Output
0034  Male  18  33  92
0066  Male  18  48  59
0092  Male  18  59  41
0115  Female  18  65  48
0001  Male  19  15  39
0062  Male  19  46  55
0069  Male  19  48  59
0112  Female  19  63  54
...
...
...
0091  Female  68  59  55
0109  Male  68  63  43
0058  Male  69  44  46
0061  Male  70  46  56
0071  Male  70  49  55
```

3.2.7.2 Cas de la fonction sortByKey()

La transformation sortByKey() est une opération applicable sur un PairRDD (paire clé/valeur) qui permet de trier les valeurs par ordre croissant ou décroissant des clés. L'exemple ci-dessous illustre l'utilisation de la fonction sortByKey() sur un PairRDD.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
```

```
data = [('X6', 200), ('X3', 500), ('X1', 250), ('X4', 150), ('X7', 420)]
rdd0 = spark.sparkContext.parallelize(data)
rdd1 = rdd0.sortByKey(ascending=True)

rddElements=rdd1.collect()
for e in rddElements:
    print(e)
```

L'objet rddo étant construit à partir d'une collection de tuples, il se présente donc comme pairRDD (clé/valeur) où la clé est le premier élément de chaque tuple et la valeur le deuxième élément de chaque tuple. L'application de la fonction sortByKey() sur rddo permet de trier les valeur selon les valeur des clés. Ci-dessous le résultat de la transformation sortByKey().

```
# Output
('X1', 250)
('X3', 500)
('X4', 150)
('X6', 200)
('X7', 420)
```

Dans l'exemple ci-dessus, la fonction sortByKey() se limite simplement à trier les clés par ordre alphabétique. C'est le comportement par défaut de la fonction. Cependant, il est possible de spécifier une fonction de tri plus élaboré en utilisant, par exemple, une fonction lambda. Pour cela, il suffit de renseigner l'option keyfunc comme dans l'exemple ci-dessous :

```
rdd1=rdd0.sortByKey(ascending=True, keyfunc=lambda k: k.lower())
print(rdd1.collect())
```

Ici, l'option keyfunc spécifiée ici permet de trier selon les clés en minuscule. Bien entendu, il est possible de spécifier n'importe quelle fonction plus ou moins complexe pour l'option keyfunc.

3.2.8 Supprimer les doublons parmi les éléments d'un RDD : la fonction distinct()

Appliquer la fonction distinct() sur un RDD ou un pair RDD permet de supprimer les éléments en doublon dans le RDD. Voir exemple ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
```

```
().getOrCreate()

# Cas d'un RDD plat
data1 = [ 200, 500, 250, 500, 200, 400, 300]
rdd1=spark.sparkContext.parallelize(data1)
# Appliquer la fonction distinct()
rdd1=rdd1.distinct()
rddElements1=rdd1.collect()
for e in rddElements1:
    print(e)

# Cas d'un Pair RDD
data2 = [('X6', 200), ('X3', 500), ('X6', 200), ('X4', 150), ('X3', 500),
('X7', 500)]
rdd2=spark.sparkContext.parallelize(data2)
# Appliquer la fonction distinct()
rdd2=rdd2.distinct()
rddElements2=rdd2.collect()
for e in rddElements2:
    print(e)
```

```
# Output 1
400
300
250
200
500
# Output 2
('X4', 150)
('X7', 500)
('X3', 500)
('X6', 200)
```

3.2.9 Faire la jointure entre deux RDDs : les fonctions join(), leftouterjoin() et cartesian()

Pour effectuer la jointure entre deux RDDs, il faut d'abord les transformer en des pairs RDDs, si cela n'est pas déjà le cas (Revoir la section : transformer un RDD plat en un pair RDD). Ainsi faire la jointure entre deux RDDs revient à faire la jointure entre deux pair RDDs. Pour rappel un pair RDD est un RDD dont les éléments se présentent sous forme de tuples (clé, valeur). C'est sur la base des clés de ces tuples que la jointure est faite. Les exemples ci-dessous montrent trois types de jointure entre deux pairs RDDs.

```
# Import de toutes les librairies utilitaires usuelles
```

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

# Création Pair RDD 1
data1 = [('X1', 200), ('X2', 500), ('X3', 200), ('X4', 150), ('X5', 500),
('X6', 500)]
rdd1=spark.sparkContext.parallelize(data1)
# Création Pair RDD 2
data2 = [('X1', "Paris"), ('X2', "New York"), ('X3', "London")]
rdd2=spark.sparkContext.parallelize(data2)

# Inner join entre Pair RDD1 et pair RDD2
print("# Output Inner join")
rdd3=rdd1.join(rdd2)
rddElements3=rdd3.collect()
for e in rddElements3:
    print(e)

# leftOuterJoin entre Pair RDD1 et pair RDD2
print("# Output leftOuterJoin")
rdd4=rdd1. leftOuterJoin(rdd2)
rddElements4=rdd4.collect()
for e in rddElements4:
    print(e)

# Jointure cartésienne entre pair RDD1 et Pair RDD2
print("# Output Cartesian join")
rdd5=rdd1.cartesian(rdd2)
rddElements5=rdd5.collect()
for e in rddElements5:
    print(e)
```

```
# Output Inner join
('X3', (200, 'London'))
('X1', (200, 'Paris'))
('X2', (500, 'New York'))
# Output leftOuterJoin
('X4', (150, None))
('X3', (200, 'London'))
('X1', (200, 'Paris'))
('X5', (500, None))
('X2', (500, 'New York'))
('X6', (500, None))
# Output Cartesian join
```

```
(('X1', 200), ('X1', 'Paris'))
(('X1', 200), ('X2', 'New York'))
(('X1', 200), ('X3', 'London'))
(('X2', 500), ('X1', 'Paris'))
(('X2', 500), ('X2', 'New York'))
(('X2', 500), ('X3', 'London'))
(('X3', 200), ('X1', 'Paris'))
(('X3', 200), ('X2', 'New York'))
(('X3', 200), ('X3', 'London'))
(('X4', 150), ('X1', 'Paris'))
(('X4', 150), ('X2', 'New York'))
(('X4', 150), ('X3', 'London'))
(('X5', 500), ('X1', 'Paris'))
(('X5', 500), ('X2', 'New York'))
(('X5', 500), ('X3', 'London'))
(('X6', 500), ('X1', 'Paris'))
(('X6', 500), ('X2', 'New York'))
(('X6', 500), ('X3', 'London'))
```

Très souvent, après avoir effectué la jointure entre les deux pairs RDD, on applique une fonction map() ou flatMap() sur le pair RDD de sortie afin de combiner les valeurs venant des deux pairs RDD d'entrée. Le RDD de sortie peut être gardé sous forme de pair RDD ou transformé en RDD plat. Revoir la section « Transformer un Pair RDD en RDD plat ».

3.3 Actions sur un RDD

Contrairement aux transformations, les actions sont des opérations dont le résultat n'est pas un RDD. Les résultats des actions sont toujours mis à la disposition de l'utilisateur au niveau du driver Spark et non au niveau des executors. Les actions les plus couramment utilisées sur un RDD sont : count(), collect(), take(), first(), etc. Les exemples ci-dessous illustrent l'utilisation des actions courantes sur un RDD.

3.3.1 Compter le nombre d'éléments d'un RDD : la fonction count()

La fonction count() renvoie le nombre d'enregistrements dans un RDD

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

# Créer le RDD
data = [3.94, 4.55, 0.8, 2.87, 4.2, 8.98, 0.96, 6.61, 8.65, 0.77, 5.18,
```

```
0.89, 8.87, 0.07, 1.05, 5.7, 8.92, 7.96, 9.54,
        1.01, 0.37, 3.24]
rdd = spark.sparkContext.parallelize(data)
# Appliquer la fonction count()
print("Nb records:" + str(rdd.count()))
```

```
# Output
Nb records:22
```

3.3.2 Récupérer le premier élément d'un RDD : la fonction first()

La fonction first() appelée sur un RDD permet de renvoyer le premier enregistrement. Voir l'exemple ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

# Créer le RDD
data = [3.94,4.55,0.8,2.87,4.2
,8.98,0.96,6.61,8.65,0.77,5.18,0.89,8.87,0.07,1.05,5.7
,8.92,7.96,9.54,1.01,0.37,3.24]
rdd=spark.sparkContext.parallelize(data)
# Récupérer le premier élément du RDD
firstRec = rdd.first()
print("First Record : "+str(firstRec))
```

```
# Output
First Record : 3.94
```

3.3.3 Extraire un nombre défini d'éléments du RDD : la fonction take()

La fonction take() permet d' extraire un nombre n d'éléments du RDD. Voir exemple ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
```

```
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
# Créer le RDD
data = [3.94,4.55,0.8,2.87,4.2
,8.98,0.96,6.61,8.65,0.77,5.18,0.89,8.87,0.07,1.05,5.7
,8.92,7.96,9.54,1.01,0.37,3.24]
rdd=spark.sparkContext.parallelize(data)
# Récupérer trois éléments du RDD
taken3 = rdd.take(3) # renvoie une liste
print(taken3)
```

```
# Output
[3.94, 4.55, 0.8]
```

3.3.4 Récupérer tout le contenu du RDD : la fonction collect()

La fonction collect() permet de convertir le RDD en une array de valeurs centralisé sur le Driver Spark. Voir exemple ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
# Créer le RDD
data = [3.94,4.55,0.8,2.87,4.2
,8.98,0.96,6.61,8.65,0.77,5.18,0.89,8.87,0.07,1.05,5.7
,8.92,7.96,9.54,1.01,0.37,3.24]
rdd=spark.sparkContext.parallelize(data)
# Récupérer tous les éléments du RDD
rddElements=rdd.collect()
print("rddElements : "+str(rddElements)) # renvoie une liste d'éléments
```

```
# Output
rddElements : [3.94, 4.55, 0.8, 2.87, 4.2, 8.98, 0.96, 6.61, 8.65, 0.77,
5.18, 0.89, 8.87, 0.07, 1.05, 5.7, 8.92, 7.96, 9.54, 1.01, 0.37, 3.24]
```

3.3.5 Enregistrer le contenu d'un RDD dans un fichier texte : la fonction saveAsTextFile()

La fonction saveAsTextFile() permet d'enregistrer le contenu d'un RDD dans un fichier texte sur HDFS. Voir l'exemple ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
# Créer le RDD
data = [3.94,4.55,0.8,2.87,4.2
,8.98,0.96,6.61,8.65,0.77,5.18,0.89,8.87,0.07,1.05,5.7
,8.92,7.96,9.54,1.01,0.37,3.24]
rdd=spark.sparkContext.parallelize(data)
# Stocker le contenu du RDD dans un seul fichier text
rdd.coalesce(1).saveAsTextFile("hdfs://nnde01/user/mk/output_data1")
```

Le répertoire est créé par Spark lors de l'écriture des données. Si le répertoire existe déjà, le traitement renvoie une erreur.

Aussi, par défaut spark génère autant de fichiers que de partitions. C'est pourquoi, pour avoir tout le contenu du RDD dans un seul fichier, on peut appliquer la fonction coalesce(1) pour réduire le RDD à une seule partition et ainsi pouvoir générer un seul fichier en sortie. La fonction coalesce() est une fonction de repartitionnement du RDD. Nous reviendrons plus détails sur les usages courants de cette fonction.

Pour info, on peut relire le fichier stocké sur hdfs en utilisant la fonction textFile() ou wholeTextFiles(). Voir les détails sur chacune de ces fonctions dans la section « Créer un RDD ».

3.4 Optimiser le traitement d'un RDD : repartitionnement, mise en cache et utilisation de variables partagées

3.4.1 Repartitionner un RDD : les fonctions repartition() et coalesce()

On peut repartitionner un RDD pour remplacer le nombre de partitions par défaut généré lors de la création du RDD. Pour cela, on peut soit utiliser la fonction repartition() soit utiliser la fonction coalesce(). La fonction repartition() permet d'augmenter ou de diminuer le nombre de partitions d'un RDD alors que la fonction coalesce() permet seulement de diminuer le nombre de partitions. La fonction coalesce() est un cas particulier de la fonction repartition(). Les exemples ci-dessous illustrent l'utilisation des deux fonctions.

3.4.1.1 Cas de la fonction repartition()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
# Créer le RDD
data = [3.94,4.55,0.8 ,2.87,4.2
,8.98,0.96,6.61,8.65,0.77,5.18,0.89,8.87,0.07,1.05,5.7
,8.92,7.96,9.54,1.01,0.37,3.24]
rdd=spark.sparkContext.parallelize(data)
print("Nb partitions:"+str(rdd.getNumPartitions()))
#Repartitions n= 4
rdd=rdd.repartition(4)
print("Nb partitions:"+str(rdd.getNumPartitions()))
#Repartitions n= 7
rdd=rdd.repartition(7)
print("Nb partitions:"+str(rdd.getNumPartitions()))
```

```
# Output
Nb partitions:56
Nb partitions:4
Nb partitions:7
```

En utilisant la fonction repartition(), on peut augmenter ou diminuer le nombre de partitions d'un RDD.

3.4.1.2 Cas de la fonction coalesce()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json
```

```
spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
# Créer le RDD
data = [3.94,4.55,0.8 ,2.87,4.2
,8.98,0.96,6.61,8.65,0.77,5.18,0.89,8.87,0.07,1.05,5.7
,8.92,7.96,9.54,1.01,0.37,3.24]
rdd=spark.sparkContext.parallelize(data)
print("Nb partitions:"+str(rdd.getNumPartitions()))
#Coalesce n= 7
rdd=rdd.coalesce(7)
print("Nb partitions:"+str(rdd.getNumPartitions()))
#Coalesce n= 4
rdd=rdd.repartition(4)
print("Nb partitions:"+str(rdd.getNumPartitions()))
```

```
# Output
Nb partitions:56
Nb partitions:7
Nb partitions:4
```

La fonction coalesce() sert à diminuer le nombre de partitions d'un RDD. Elle ne permet pas de l'augmenter.

3.4.2 Mettre en cache ou persistance du RDD : les fonction cache() et persist()

La mise en cache ou la persistance d'un RDD sont des techniques d'optimisation qui visant à améliorer la performance d'exécution des traitements du RDD. La mise en cache et la persistance sont faites avec les fonctions cache() et persist(). Ces fonctions permettent de stocker sur la mémoire JVM les résultats des transformations précédemment spécifiées. Ces résultats intermédiaires sont ensuite utilisés dans la suite du traitement sans avoir à recalculer le RDD initial. En effet, nous savons que toutes les opérations de transformation d'un RDD sont « lazzy », c'est-à-dire que les transformations ne s'exécutent que lorsqu'une action est appelée. Rappelons cependant que chaque appel d'une action sur un RDD recompile toutes les transformations spécifiées en amont pour calculer ce RDD. La persistance et la mise en cache du RDD sont des moyens qui permettent d'éviter ces opérations de recompilation, car ils permettent d'exécuter toutes les transformations en amont et de stocker les résultats sur la mémoire de travail. Ces résultats sont ainsi directement utilisables par les transformations et les actions suivantes sans avoir à recommencer les opérations de traitement depuis le début. La persistance et la mise en cache apparaissent ainsi des techniques d'optimisation des traitements spark. L'exemple ci-dessous illustre l'utilisation des fonctions cache() et persist().

3.4.2.1 Cas de la fonction cache()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [3.94,4.55,0.8 ,2.87,4.2
,8.98,0.96,6.61,8.65,0.77,5.18,0.89,8.87,0.07,1.05,5.7
,8.92,7.96,9.54,1.01,0.37,3.24]
rdd=spark.sparkContext.parallelize(data)

#Transformation 1
rdd1=rdd.map(lambda x: x*10) #multiplier chaque élément par 10
#Transformation 2
rdd2=rdd1.filter(lambda x : x>10) #Retenir tous les éléments > 10
#Mise en cache()
rdd2.cache()
#Action1
print("Nb records:"+str(rdd2.count()))
#Action2
elems = rdd2.collect()
print("elems:"+ str(elems))
```

```
# Output
Nb records:16
elems:[39.4, 45.5, 28.70, 42.0, 89.80, 66.100, 86.5, 51.8, 88.69, 10.5,
57.0, 89.2, 79.6, 95.39, 10.1, 32.40]
```

3.4.2.2 Cas de la fonction persist()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [3.94,4.55,0.8 ,2.87,4.2
,8.98,0.96,6.61,8.65,0.77,5.18,0.89,8.87,0.07,1.05,5.7
```

```
,8.92,7.96,9.54,1.01,0.37,3.24]
rdd=spark.sparkContext.parallelize(data)

#Transformation 1
rdd1=rdd.map(lambda x: x*10) #multiplier chaque élément par 10
#Transformation 2
rdd2=rdd1.filter(lambda x : x>10) #Retenir tous les éléments > 10
#Persist
rdd2.persist()
#Action1
print("Nb records:"+str(rdd2.count()))
#Action2
elems = rdd2.collect()
print("elems:"+ str(elems))
#Unpersist
rdd2.unpersist()
```

```
# Output
Nb records:16
elems:[39.4, 45.5, 28.70, 42.0, 89.80, 66.100, 86.5, 51.8, 88.69, 10.5,
57.0, 89.2, 79.6, 95.39, 10.1, 32.40]
```

La fonction persist() peut être accompagnée par une fonction unpersist() qui permet de dé-persister le RDD dès que l'on a terminé les opérations de traitement sur le RDD. A noter aussi que la fonction unpersist() peut être utilisé à l a suite d'une fonction cache(). Dans ce cas, elle permet de dé-cacher le RDD.

Niveau de mise en cache et de persistance

Lorsqu'on utilise les fonctions cache() ou persist(), les résultats des calculs sont stockés par défaut sur la mémoire de travail. Cependant, ce comportement par défaut est insuffisant dans de nombreuses situations comme par exemple lorsque la taille du RDD dépasse la taille de la mémoire. Mais Spark propose différentes options pour le stockage des résultats aussi bien sur la mémoire que sur le disque avec de multiples variantes. Les options présentées ci-dessous représentent les différents niveau de stockage pouvant être utilisés en options des fonctions cache() et persist().

- **MEMORY_ONLY** : Stocke le RDD sur la mémoire uniquement. C'est l'option par défaut de la fonction cache(). On peut encore spécifier cache(MEMORY_ONLY).
- **MEMORY_ONLY_2** : Similaire à l'option MEMORY_ONLY mais réplique chaque partition sur deux executors distincts.
- **MEMORY_ONLY_SER** : stocke le RDD uniquement sur la mémoire mais sous forme d'objets sérialisés. La sérialisation du RDD permet de prendre moins d'espace que l'option MEMORY_ONLY.

- **MEMORY_ONLY_SER_2** : Similaire à l'option MEMORY_ONLY_SER mais réplique chaque partition sur deux executors distincts.
- **MEMORY_AND_DISK** : Stocke une partie du RDD sur la mémoire JVM et l'excédent sur le disque lorsque la taille de la mémoire n'est pas suffisante pour contenir tout le RDD.
- **MEMORY_AND_DISK_2** : Identique à l'option MEMORY_AND_DISK mais réplique chaque partition sur deux executors distincts.
- **MEMORY_AND_DISK_SER** : Similaire à l'option MEMORY_AND_DISK mais stocke le RDD sous formes d'objets sérialisés.
- **MEMORY_AND_DISK_SER_2** : Similaire à l'option MEMORY_AND_DISK_SER mais réplique chaque partition sur deux executors distincts.
- **DISK_ONLY** : Stocke le RDD uniquement sur le disque. Cette option nécessite beaucoup plus d'opération E/S lors du traitement du RDD.
- **DISK_ONLY_2** : Similaire à l'option DISK_ONLY mais réplique chaque partition sur deux executors distincts.

3.4.3 Utiliser les variables partagées sur un RDD : les fonctions broadcast() et accumulator()

Deux variables partagées sont couramment utilisées dans les traitements Spark, il s'agit notamment des variables broadcastées et des variables accumulées.

Les variables broadcastées sont des variables envoyées à chaque executor afin faciliter les opérations de traitement des partitions du RDD situés sur cet executor. Les variables accumulées sont des variables permettant d'obtenir des informations agrégées sur le RDD sans avoir besoin d'appliquer une action : comptage du nombre d'éléments d'un RDD, somme d'un attribut, etc.

Les exemples ci-dessous illustrent les cas d'utilisation de la fonction broadcast() et de la fonction accumulator().

Alors que les variables accumulées sont obtenues en utilisant la fonction accumulator().

3.4.3.1 Cas de la fonction broadcast()

Le broadcast consiste, par exemple, à déposer une copie d'un dataset dans chaque executor près des partitions du RDD à traiter. Le dépôt de la copie du dataset dans l'executor assure la data locality et évite les multiples transferts de données via le réseau (shuffle) qui sont parfois très couteux. Dans l'exemple ci-dessous, nous déposons une copie de la collection data sur chaque executor du Job Spark.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
```

```
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

from pyspark.sql import SparkSession
spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = ["New York", "California", "Florida"]
bc = spark.sparkContext.broadcast(data)
print(bc.value)
# N'importe quelle type de données peut être broadcasté: rdd, collection,
etc.
```

```
# Output
['New York', 'California', 'Florida']
```

Dans l'exemple ci-dessus, nous avons broadcasté une collection, mais il est possible de broadcaster n'importe quel type de données y compris un autre RDD.

3.4.3.2 Cas de la fonction accumulator()

Une accumulator renvoie une information agrégée sur un RDD. Les exemples ci-dessus illustrent l'utilisation de la fonction accumulator() pour calculer le nombre d'éléments et la somme des valeurs des éléments d'un RDD.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [3.94,4.55,0.8,2.87,4.2
,8.98,0.96,6.61,8.65,0.77,5.18,0.89,8.87,0.07,1.05,5.7
,8.92,7.96,9.54,1.01,0.37,3.24]
rdd=spark.sparkContext.parallelize(data)

#Accumulator de comptage des éléments du rdd
ac1=spark.sparkContext.accumulator(0)
rdd.foreach(lambda x:ac1.add(1))
print(ac1.value) #renvoie la valeur finale accumulée.
```

```
#Accumulator de somme des éléments du rdd
ac2= spark.sparkContext.accumulator(0)
rdd.foreach(lambda x:ac2.add(x))
print(ac2.value) #renvoie la valeur finale accumulée.
```

```
# Output ac1
22
# Output ac2
95.13
```

4 PYSPARK DATAFRAME

Le dataframe PySpark est une collection de données distribuée ayant un schéma bien défini, c'est-à-dire une collection de données organisées en lignes et en colonnes dont les types sont connus à l'avance. La structure d'un dataframe PySpark est, conceptuellement, équivalent à celle d'une table standard d'une base de données relationnelle ou à celle d'un dataframe Python/R. Cependant, le dataframe PySpark bénéficie de beaucoup plus d'optimisation qui expliquent sa performance de traitement par rapport à la plupart des structures précédemment évoqués.

Dans les sections qui vont suivre, nous allons passer en revue quelques opérations de traitement applicables sur un dataframe lors d'un process de data engineering.

4.1 Créer un Dataframe : la fonction createDataFrame()

4.1.1 Construire un dataframe à partir d'une collection de données : la fonction createDataFrame()

La fonction createDataFrame() est la fonction de base servant à créer un dataframe PySpark. L'exemple ci-dessous illustre la création d'un dataframe à partir d'une collection.

4.1.1.1 Collection contenant plusieurs colonnes

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json


spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
# Afficher le schema du dataframe
df.printSchema()
```

```
# Afficher le contenu du Dataframe
df.show(truncate=False)
```

```
# Output printSchema()
|-- prenom: string (nullable = true)
|-- surnom: string (nullable = true)
|-- nom: string (nullable = true)
|-- date_naiss: string (nullable = true)
|-- genre: string (nullable = true)
|-- salaire: long (nullable = true)

# Output show()
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+
```

La collection servant à créer le dataframe est toujours spécifiée sous forme d'une liste de tuples () ; Chaque tuple() représentera une ligne dans le dataframe obtenu en sortie. Pour chaque tuple(), les valeurs des colonnes sont séparées par une virgule.

4.1.1.2 Collection contenant une seule colonne

Lorsqu'il s'agit de créer un dataframe avec une seule colonne, la valeur unique spécifiée dans le tuple () doit être suivie par une virgule. Ex : (230,) pour une ligne donnée. Voir exemple ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json


spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

# Créer un Dataframe à une seule colonne
data = [    ('Pascal',),
            ('Pierre',),
            ('Laurent',),
            ('Virginie',),
```

```
            ('Jenny',)
    ]

columns = ["prenom"]
df = spark.createDataFrame(data=data, schema = columns)
df.printSchema()
df.show(truncate=False)
```

```
# Output printSchema() df
| -- prenom: string(nullable=true)

# Output show() df
+--------+
|prenom  |
+--------+
|Pascal  |
|Pierre  |
|Laurent |
|Virginie|
|Jenny   |
+--------+
```

Dans l'exemple ci-dessus, nous avons créé un dataframe à une seule colonne contenant des prénoms.

4.1.2 Définir le schéma d'un dataframe : les fonctions Structype(), StructField() et StructType.fromJson

Définir le schéma d'un dataframe consiste à indiquer les colonnes ainsi que leur type. D'une manière générale, on définit le schéma lors de la création d'un dataframe. Pour cela, on utilise les fonctions StructType et StructField. La fonction StructType permet d'indiquer la liste des colonnes ainsi que leur type. StructField permet de spécifier individuellement le type de chaque colonne faisant partie de la liste. Quant à la fonction StructType.fromJson, elle permet de définir le schéma du dataframe en se basant sur des spécifications définies dans un fichier json. Les exemples qui vont suivre illustrent différents cas d'utilisation des fonctions Structype() et StructField() mais également la fonction StructType.fromJson

4.1.2.1 Cas d'un schéma à structure simple

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json
```

```
spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]

schema_df = StructType([ \
    StructField("prenom",StringType(),True), \
    StructField("surnom",StringType(),True), \
    StructField("nom",StringType(),True), \
    StructField("date_naiss", StringType(), True), \
    StructField("genre", StringType(), True), \
    StructField("salaire", LongType(), True) \
  ])

df = spark.createDataFrame(data=data, schema = schema_df)
df.printSchema()
df.show(truncate=False)
```

```
# Output printSchema()
| -- prenom: string(nullable=true)
| -- surnom: string(nullable=true)
| -- nom: string(nullable=true)
| -- date_naiss: string(nullable=true)
| -- genre: string(nullable=true)
| -- salaire: long(nullable=true)

# Output show()
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+
```

La liste des types Spark est disponible sur ce lien :
https://spark.apache.org/docs/latest/sql-ref-datatypes.html

4.1.2.2 Cas d'un schéma à structure complexe : schéma imbriqué

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
```

```
from pyspark.sql.window import Window
import json


spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

structureData = [
    (("Pascal","","Martin"),"36636","M",3100),
    (("Pierre","Rose",""),"40288","M",4300),
    (("Laurent","","Bernard"),"42114","M",1400),
    (("Virginie","Anne","Carpentier"),"39192","F",5500),
    (("Jenny","Carole","Bouvier"),"","F",3500)
  ]
structureSchema = StructType([
        StructField('nom_complet', StructType([
             StructField('prenom', StringType(), True),
             StructField('surnom', StringType(), True),
             StructField('nom', StringType(), True)
             ])),
         StructField('id', StringType(), True),
         StructField('genre', StringType(), True),
         StructField('salaire', LongType(), True)
         ])

df2 = spark.createDataFrame(data=structureData,schema=structureSchema)
df2.printSchema()
df2.show(truncate=False)
```

```
# Output printSchema()
| -- nom_complet: struct(nullable=true)
| | -- prenom: string(nullable=true)
| | -- surnom: string(nullable=true)
| | -- nom: string(nullable=true)
| -- id: string(nullable=true)
| -- genre: string(nullable=true)
| -- salaire: long(nullable=true)

# Output show()
+-----------------------------+-----+-----+-------+
|nom_complet                  |id   |genre|salaire|
+-----------------------------+-----+-----+-------+
|[Pascal, , Martin]           |36636|M    |3100   |
|[Pierre, Rose, ]             |40288|M    |4300   |
|[Laurent, , Bernard]         |42114|M    |1400   |
|[Virginie, Anne, Carpentier]|39192|F    |5500   |
|[Jenny, Carole, Bouvier]     |     |F    |3500   |
+-----------------------------+-----+-----+-------+
```

4.1.2.3 Construire le schéma d'un Dataframe à partir d'un fichier json de métadonnées : la fonction StructType.fromJson()

L'exemple ci-dessous montre la construction du schéma d'un Dataframe à partir d'un fichier json en utilisant la fonction StructType.fromJson()

Soit meta.json le fichier contenant la spécification suivante des métadonnées (colonnes et leur type). Le contenu du fichier se présente comme suit :

```
{
  "type" : "struct",
  "fields" : [ {
    "name" : "nom_complet",
    "type" : {
      "type" : "struct",
      "fields" : [ {
        "name" : "prenom",
        "type" : "string",
        "nullable" : true,
        "metadata" : { }
      }, {
        "name" : "surnom",
        "type" : "string",
        "nullable" : true,
        "metadata" : { }
      }, {
        "name" : "nom",
        "type" : "string",
        "nullable" : true,
        "metadata" : { }
      } ]
    },
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "date_naiss",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "genre",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "salaire",
    "type" : "integer",
    "nullable" : true,
    "metadata" : { }
  } ]
}
```

On peut construire le schéma d'un dataframe à partir de ce fichier en suivant la procédure décrite dans le code ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
```

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

structureData = [
    (("Pascal","","Martin"),"36636","M",3100),
    (("Pierre","Rose",""),"40288","M",4300),
    (("Laurent","","Bernard"),"42114","M",1400),
    (("Virginie","Anne","Carpentier"),"39192","F",5500),
    (("Jenny","Carole","Bouvier"),"","F",3500)
  ]
rdd= spark.sparkContext.parallelize(structureData)

def readFile(jsonFile):
    with open(jsonFile) as js_content:
        metadata = json.load(js_content)
    return metadata

schemaFromJson = StructType.fromJson(readFile("meta.json"))
df = spark.createDataFrame(rdd,schemaFromJson)
df.printSchema()
df.show()
```

```
# Output printSchema()
| -- nom_complet: struct(nullable=true)
| | -- prenom: string(nullable=true)
| | -- surnom: string(nullable=true)
| | -- nom: string(nullable=true)
| -- date_naiss: string(nullable=true)
| -- genre: string(nullable=true)
| -- salaire: integer(nullable=true)

# Output show()
+----------------------------+----------+-----+-------+
|nom_complet                 |date_naiss|genre|salaire|
+----------------------------+----------+-----+-------+
|[Pascal, , Martin]          |36636     |M    |3100   |
|[Pierre, Rose, ]            |40288     |M    |4300   |
|[Laurent, , Bernard]        |42114     |M    |1400   |
|[Virginie, Anne, Carpentier]|39192     |F    |5500   |
|[Jenny, Carole, Bouvier]    |          |F    |3500   |
+----------------------------+----------+-----+-------+
```

Pour pouvoir construire le schéma à partir d'un fichier, celui-ci doit être ajouté en tant que fichier ressource lors du lancement du job spark. Nous reviendrons plus tard sur les configurations nécessaires pour le lancement d'un traitement Spark.

4.1.3 Créer un dataframe vide : la fonction emptyRDD() ou le pseudo filtre filter(F.lit(1)==F.lit(2))

4.1.3.1 Créer un dataframe vide à partir d'un RDD vide en spécifiant le schema

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

emptyRDD = spark.sparkContext.emptyRDD()

columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
schema = StructType([
  StructField(columns[0], StringType(), True),
  StructField(columns[1], StringType(), True),
  StructField(columns[2], StringType(), True),
  StructField(columns[3], DateType(), True),
  StructField(columns[4], StringType(), True),
  StructField(columns[5], DoubleType(), True)
  ])

emptyDf = spark.createDataFrame(emptyRDD,schema)
print(emptyDf.printSchema())
print("Nb rows :"+str(emptyDf.count()))
```

```
# Output printSchema()
root
 |-- prenom: string (nullable = true)
 |-- surnom: string (nullable = true)
 |-- nom: string (nullable = true)
 |-- date_naiss: date (nullable = true)
 |-- genre: string (nullable = true)
 |-- salaire: double (nullable = true)

# Output count()
Nb rows :0
```

4.1.3.2 Créer un dataframe vide à partir d'un autre dataframe en utilisant un pseudo filtre filter(F.lit(1)==F.lit(2))

Techniquement, on peut créer un dataframe vide à partir d'un autre dataframe dont le schéma est déjà connu. Pour cela, il suffit d'appliquer un filtre dont les conditions ne

seront pas satisfaites. Cette méthode s'appelle pseuso-filtre. L'exemple ci-dessous illustre la création d'un dataframe vide en utilisant un pseudo-filtre.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.printSchema()
df.show(truncate=False)

emptyDf=df.filter(F.lit(1)==F.lit(2)) # Créer un dataframe vide en utiliant
un pseudo-filter()
emptyDf.show()
emptyDf.printSchema()
if emptyDf.rdd.isEmpty():
        print("df is empty")
```

```
#output printSchema() df

root
 |-- prenom: string (nullable = true)
 |-- surnom: string (nullable = true)
 |-- nom: string (nullable = true)
 |-- date_naiss: string (nullable = true)
 |-- genre: string (nullable = true)
 |-- salaire: long (nullable = true)

# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+
```

```
#output printSchema() df

root
 |-- prenom: string (nullable = true)
 |-- surnom: string (nullable = true)
 |-- nom: string (nullable = true)
 |-- date_naiss: string (nullable = true)
 |-- genre: string (nullable = true)
 |-- salaire: long (nullable = true)

# Output show() df
+------+------+---+----------+-----+-------+
|prenom|surnom|nom|date_naiss|genre|salaire|
+------+------+---+----------+-----+-------+
+------+------+---+----------+-----+-------+

# Output print text:
df is empty
```

4.1.4 Créer un Dataframe à partir d'un RDD et créer un RDD à partir d'un Dataframe : la fonction toDF() et la fonction rdd()

Les exemples ci-dessous illustrent la création d'un Dataframe à partir d'un RDD ainsi que la création d'un RDD à partir d'un Dataframe.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

################## Cas du Dataframe à une seule colonne
#Construction du RDD
data1 = [(3.94,)
        ,(4.55,)
        ,(0.8,)
        ,(2.87,)
        ,(4.2,)
        ,(8.98,)
        ,(0.96,)
        ,(6.61,)
        ,(8.65,)
            ]
rdd1=spark.sparkContext.parallelize(data1)

# Création du Dataframe à partir du RDD
col_names1=["prix"]
df1 = rdd1.toDF(col_names1)
df1.printSchema()
```

```
df1.show(truncate=False)
# # Création du RDD à partir du Dataframe
rdd1 = df1.rdd
print(rdd1.collect())
for e in rdd1.collect():
    print(e["prix"])


################## Cas du Dataframe à plusieurs colonnes
#Construction du RDD
data2 = [(3.94, 2)
        ,(4.55, 6)
        ,(0.8, 1)
        ,(2.87, 8)
        ,(4.2,4)
        ,(8.98, 2)
        ,(0.96,10)
        ,(6.61, 5)
        ,(8.65,6)
            ]
rdd2=spark.sparkContext.parallelize(data2)
# Création du Dataframe à partir du RDD
col_names2=["prix", "quantite"]
df2 = rdd2.toDF(col_names2)
df2.printSchema()
df2.show(truncate=False)

# # Création du RDD à partir du Dataframe
rdd2 = df2.rdd
print(rdd2.collect())
for e in rdd2.collect():
    print(e["prix"],";",e["quantite"])
```

```
# Output printSchema() df1
| -- prix: double(nullable=true)
# Output show() df1
+----+
|prix|
+----+
|3.94|
|4.55|
|0.8 |
|2.87|
|4.2 |
|8.98|
|0.96|
|6.61|
|8.65|
+----+
# Output collect() rdd1
[Row(prix=3.94), Row(prix=4.55), Row(prix=0.8), Row(prix=2.87),
Row(prix=4.2), Row(prix=8.98), Row(prix=0.96), Row(prix=6.61),
Row(prix=8.65)]

# Output print des éléments extraits de rdd1
3.94
4.55
```

```
0.8
2.87
4.2
8.98
0.96
6.61
8.65


# Output printSchema() df2
| -- prix: double(nullable=true)
| -- quantite: long(nullable=true)
# Output show() df2
+----+--------+
|prix|quantite|
+----+--------+
|3.94|2       |
|4.55|6       |
|0.8 |1       |
|2.87|8       |
|4.2 |4       |
|8.98|2       |
|0.96|10      |
|6.61|5       |
|8.65|6       |
+----+--------+
# Output collect() rdd2
[Row(prix=3.94, quantite=2), Row(prix=4.55, quantite=6), Row(prix=0.8,
quantite=1), Row(prix=2.87, quantite=8), Row(prix=4.2, quantite=4),
Row(prix=8.98, quantite=2), Row(prix=0.96, quantite=10), Row(prix=6.61,
quantite=5), Row(prix=8.65, quantite=6)]

# Output print des éléments extraits de rdd2
3.94 ; 2
4.55 ; 6
0.8 ; 1
2.87 ; 8
4.2 ; 4
8.98 ; 2
0.96 ; 10
6.61 ; 5
8.65 ; 6
```

4.2 Créer un dataframe à partir des fichiers de données : csv, txt, json et parquet

4.2.1.1 Créer le dataframe à partir d'un fichier csv : la fonction read.csv()

PySpark offre plusieurs fonctionnalités de lecture des fichiers csv. Il s'agit notamment des fonctions read.csv() et read.format("csv").load(). Les exemples ci-dessous montrent le cas particulier de l'utilisation de la fonction read.csv().

Soit un fichier csv stocké sur hdfs et dont le contenu se présente comme ceci.

movie_rating1.csv

```
"userId","movieId","rating","ts"
1,2,3.5,2005-04-02 23:53:47
2,29,3.5,2005-04-02 23:31:16
3,32,3.5,2005-04-02 23:33:39
4,47,3.5,2005-04-02 23:32:07
5,50,3.5,2005-04-02 23:29:40
```

Le bout code ci-dessous permet de créer un dataframe à partir de ce fichier.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

df = spark.read.options(
            delimiter=","
            ,header=True
            ,quote ="\""
            , nullValue =None
            ,nanValue ="NaN"
            ,inferSchema =True
            ,encoding ="utf-8"
        ).csv("hdfs://nnde01/user/mk/csv_data/input/movie_rating1.csv")

df.printSchema()
df.show()
```

```
# Output printSchema()
| -- userId: integer(nullable=true)
| -- movieId: integer(nullable=true)
| -- rating: double(nullable=true)
| -- ts: timestamp(nullable=true)
# Output show()
+------+-------+------+-------------------+
|userId|movieId|rating|                 ts|
+------+-------+------+-------------------+
|     1|      2|   3.5|2005-04-02 23:53:47|
|     2|     29|   3.5|2005-04-02 23:31:16|
|     3|     32|   3.5|2005-04-02 23:33:39|
|     4|     47|   3.5|2005-04-02 23:32:07|
|     5|     50|   3.5|2005-04-02 23:29:40|
+------+-------+------+-------------------+
```

Plusieurs options peuvent être ajoutées à la fonction read.csv() afin de prendre en compte les propriétés spécifiques au csv. Le tableau ci-dessous présente les principales options pouvant être ajoutées à la fonction read.csv().

Options	Description
header	Indique si le csv contient une entête, c'est-à-dire le nom des colonnes
delimiter	Permet de spécifier le délimiteur du csv
quote	Permet d'indiquer le caractère qui représente les quotes dans les données : " ou '.
nullValue	Permet d'indiquer le caractère qui représente les valeurs nulles dans le csv
nanValue	Permet d'indiquer le caractère qui représente les valeurs NaN (not a number) dans le csv
inferSchema	Indique à Spark de déduire (deviner) le type de chaque colonne du csv
encoding	Permet d'indiquer l'encoding du fichier. Ex : utf-8, etc..

Par ailleurs, notons aussi que le path spécifié dans la fonction .csv() peut être aussi bien un fichier csv ciblé qu'un répertoire complet. Par exemple, on peut spécifier soit : csv("/hdfs_directory/source/data.csv") ou spécifier csv("/hdfs_directory/source/"). Cependant lorsque le path spécifié est un répertoire, il faut bien s'assurer que les fichiers que ce répertoire contient sont tous des csv de même format et de même structure.

4.2.1.2 Créer le dataframe à partir d'un fichier texte plat : la fonction read.txt()

Soit un fichier texte stocké sur hdfs dont le contenu se présente comme ceci.

movie_ref.txt

```
1;Toy Story (1995);Adventure|Animation|Children|Comedy|Fantasy
2;Jumanji (1995);Adventure|Children|Fantasy
3;Grumpier Old Men (1995);Comedy|Romance
4;Waiting to Exhale (1995);Comedy|Drama|Romance
5;Father of the Bride Part II (1995);Comedy
6;Heat (1995);Action|Crime|Thriller
7;Sabrina (1995);Comedy|Romance
8;Tom and Huck (1995);Adventure|Children
9;Sudden Death (1995);Action
10;GoldenEye (1995);Action|Adventure|Thriller
11;American President, The (1995);Comedy|Drama|Romance
12;Dracula: Dead and Loving It (1995);Comedy|Horror
13;Balto (1995);Adventure|Animation|Children
14;Nixon (1995);Drama
15;Cutthroat Island (1995);Action|Adventure|Romance
16;Casino (1995);Crime|Drama
17;Sense and Sensibility (1995);Drama|Romance
18;Four Rooms (1995);Comedy
19;Ace Ventura: When Nature Calls (1995);Comedy
20;Money Train (1995);Action|Comedy|Crime|Drama|Thriller
21;Get Shorty (1995);Comedy|Crime|Thriller
```

On peut créer un dataframe à partir de ce fichier en utilisant la fonction read.txt() telle que présentée dans l'exemple ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

## Construire le dataframe brut à partir du fichier txt
raw_df = spark.read.text("hdfs://nnde01/user/mk
/txt_data/input/movie_ref.txt") # d'autres argument ,lineSep= : '\n',
'\r','\r\n'
raw_df.printSchema()
raw_df.show(truncate=False)

# Construire le dataframe formaté
final_df=raw_df.select(F.split(F.col("value"),";").getItem(0).alias("movieI
d")
,F.split(F.col("value"),";").getItem(1).alias("title")
,F.split(F.col("value"),";").getItem(2).alias("genres"))

final_df.printSchema()
final_df.show(truncate=False)
```

```
# Output printSchema() raw df
| -- value: string(nullable=true)

# Output show() raw_df
+------------------------------------------------------------------+
|value                                                             |
+------------------------------------------------------------------+
|1;Toy Story (1995);Adventure|Animation|Children|Comedy|Fantasy|
|2;Jumanji (1995);Adventure|Children|Fantasy                      |
|3;Grumpier Old Men (1995);Comedy|Romance                          |
|4;Waiting to Exhale (1995);Comedy|Drama|Romance                   |
|5;Father of the Bride Part II (1995);Comedy                       |
|6;Heat (1995);Action|Crime|Thriller                              |
|7;Sabrina (1995);Comedy|Romance                                  |
|8;Tom and Huck (1995);Adventure|Children                         |
...
...
...

# Output printSchema() final df
| -- movieId: string(nullable=true)
| -- title: string(nullable=true)
| -- genres: string(nullable=true)

# Output show() final_df
+-------+-----------------------------------+-----------------------------------------+
|movieId|title                              |genres                                   |
+-------+-----------------------------------+-----------------------------------------+
|1      |Toy Story (1995)                   |Adventure|Animation|Children|Comedy|Fantasy|
|2      |Jumanji (1995)                     |Adventure|Children|Fantasy                |
|3      |Grumpier Old Men (1995)            |Comedy|Romance                           |
|4      |Waiting to Exhale (1995)           |Comedy|Drama|Romance                    |
```

```
|5      |Father of the Bride Part II (1995)    |Comedy                                |
|6      |Heat (1995)                           |Action|Crime|Thriller                 |
|7      |Sabrina (1995)                        |Comedy|Romance                        |
|8      |Tom and Huck (1995)                   |Adventure|Children                    |
...
...
...
```

4.2.1.3 Créer le dataframe à partir d'un fichier json : la fonction read.json()

Cas d'un fichier json plat : le json sur une seule ligne

Soit un fichier json stocké sur hdfs et dont le contenu se présente comme suit :

employees1.json

```
{"prenom":"Pascal","surnom":"","nom":"Martin","date_naiss":"1995-02-25","genre":"M","salaire":4500}
{"prenom":"Pierre","surnom":"Rose","nom":"","date_naiss":"1998-07-12","genre":"M","salaire":3000}
{"prenom":"Laurent","surnom":"","nom":"Bernard","date_naiss":"2000-09-14","genre":"M","salaire":3000}
{"prenom":"Virginie","surnom":"Anne","nom":"Carpentier","date_naiss":"1989-10-06","genre":"F","salaire":2500}
{"prenom":"Jenny","surnom":"Carole","nom":"Bouvier","date_naiss":"1982-02-15","genre":"F","salaire":3500}
```

L'exemple ci-dessous montre l'utilisation de la fonction read.json() pour créer un dataframe à partir de ce fichier json.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

df =
spark.read.option("multiline","false").json("hdfs://nnde01/user/mk/json_dat
a/input/employees1.json")
df.printSchema()
df.show(truncate=False)
```

```
# Output printSchema() df
| -- date_naiss: string(nullable=true)
| -- genre: string(nullable=true)
| -- nom: string(nullable=true)
| -- prenom: string(nullable=true)
| -- salaire: long(nullable=true)
```

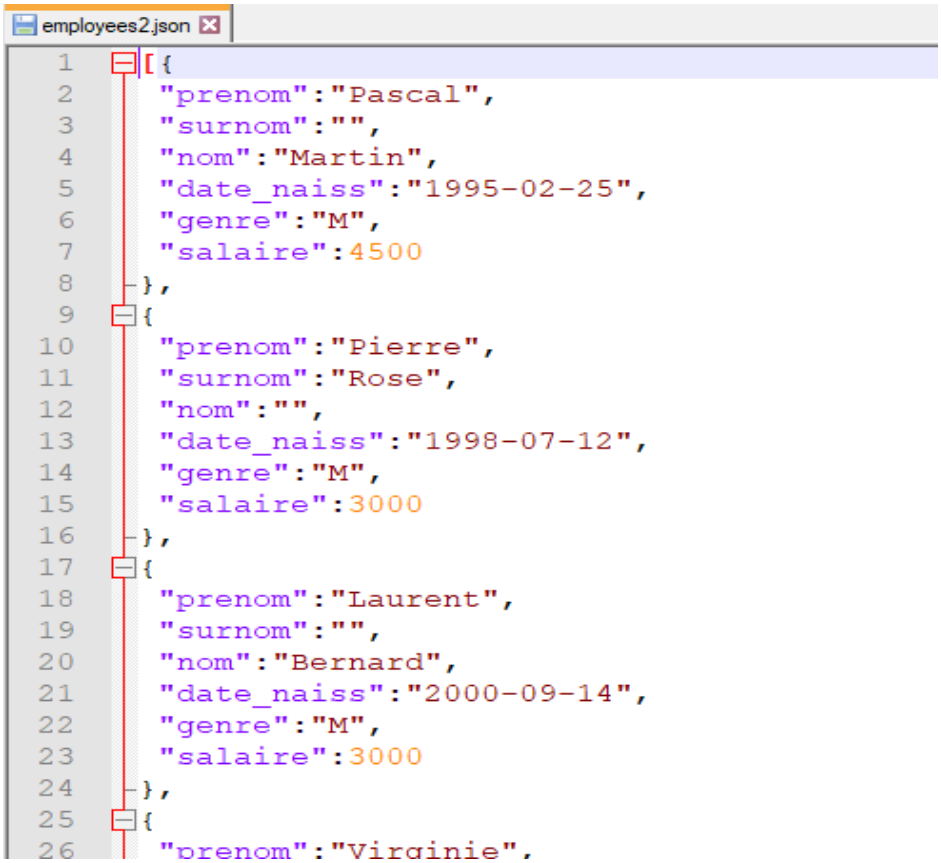
```
| -- surnom: string(nullable=true)

# Output show() df
+----------+-----+----------+--------+-------+------+
|date_naiss|genre|nom       |prenom  |salaire|surnom|
+----------+-----+----------+--------+-------+------+
|1995-02-25|M    |Martin    |Pascal  |4500   |      |
|1998-07-12|M    |          |Pierre  |3000   |Rose  |
|2000-09-14|M    |Bernard   |Laurent |3000   |      |
|1989-10-06|F    |Carpentier|Virginie|2500   |Anne  |
|1982-02-15|F    |Bouvier   |Jenny   |3500   |Carole|
+----------+-----+----------+--------+-------+------+
```

On peut lire plusieurs fichiers json dans un même répertoire. Cependant, il faut s'assurer que les fichiers json présents ont la même structure. La ligne de code ci-dessous montre comment spécifier la lecture plusieurs fichiers jsons.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

df =
spark.read.option("multiline","false").json("hdfs://nnde01/user/mk/json_dat
a/input/*.json")
df.printSchema()
df.show(truncate=False)
```

Cas d'un fichier json plat : le json s'étend sur plusieurs lignes

Dans l'exemple précédent, chaque élément json se trouve sur une seule ligne du fichier. Mais il arrive qu'un élément json s'étend sur plusieurs lignes du fichier comme le fichier ci-dessous. Dans ce genre de situation, il faut mettre l'option multiline à true.



L'exemple ci-dessous illustre comment créer un dataframe à partir de ce fichier.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

df =
spark.read.option("multiline","true").json("hdfs://nnde01/user/mk/json_data
/input/employees2.json")
df.printSchema()
df.show(truncate=False)
```

```
# Output printSchema() df
| -- date_naiss: string(nullable=true)
| -- genre: string(nullable=true)
| -- nom: string(nullable=true)
| -- prenom: string(nullable=true)
| -- salaire: long(nullable=true)
| -- surnom: string(nullable=true)
```

```
# Output show() df
+----------+-----+----------+--------+-------+------+
|date naiss|genre|nom       |prenom  |salaire|surnom|
+----------+-----+----------+--------+-------+------+
|1995-02-25|M    |Martin    |Pascal  |4500   |      |
|1998-07-12|M    |          |Pierre  |3000   |Rose  |
|2000-09-14|M    |Bernard   |Laurent |3000   |      |
|1989-10-06|F    |Carpentier|Virginie|2500   |Anne  |
|1982-02-15|F    |Bouvier   |Jenny   |3500   |Carole|
+----------+-----+----------+--------+-------+------+
```

Cas d'un fichier json imbriqué (nested json) : le json sur une seule ligne

Soit le fichier json stocké sur hdfs. Les éléments json dans le fichier sont imbriqués et chaque élément json se trouve sur une seule ligne.

employees3.json

```
{"nom_complet":{"prenom":"Pascal","surnom":"","nom":"Martin"},"date_naiss":"1995-02-25","genre":"M","salaire":4500}
{"nom_complet":{"prenom":"Pierre","surnom":"Rose","nom":""},"date_naiss":"1998-07-12","genre":"M","salaire":3000}
{"nom_complet":{"prenom":"Laurent","surnom":"","nom":"Bernard"},"date_naiss":"2000-09-14","genre":"M","salaire":3000}
{"nom_complet":{"prenom":"Virginie","surnom":"Anne","nom":"Carpentier"},"date_naiss":"1989-10-06","genre":"F","salaire":2500}
{"nom_complet":{"prenom":"Jenny","surnom":"Carole","nom":"Bouvier"},"date_naiss":"1982-02-15","genre":"F","salaire":3500}
```

L'exemple ci-dessous montre comment créer un dataframe à partir de ce fichier.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

df =
spark.read.option("multiline","false").json("hdfs://nnde01/user/mk/json_dat
a/input/employees3.json")
df.printSchema()
df.show(truncate=False)
```

```
| -- date_naiss: string(nullable=true)
| -- genre: string(nullable=true)
| -- nom_complet: struct(nullable=true)
| | -- nom: string(nullable=true)
| | -- prenom: string(nullable=true)
| | -- surnom: string(nullable=true)
| -- salaire: long(nullable=true)
```

```
+----------+-----+--------------------------+-------+
|date_naiss|genre|nom_complet               |salaire|
+----------+-----+--------------------------+-------+
|1995-02-25|M    |[Martin, Pascal, ]        |4500   |
|1998-07-12|M    |[, Pierre, Rose]          |3000   |
|2000-09-14|M    |[Bernard, Laurent, ]      |3000   |
|1989-10-06|F    |[Carpentier, Virginie, Anne]|2500   |
|1982-02-15|F    |[Bouvier, Jenny, Carole]  |3500   |
+----------+-----+--------------------------+-------+
```

Cas d'un fichier json imbriqué (nested json) : le json s'étend sur plusieurs lignes

Soit le fichier json stocké sur hdfs. Les éléments json dans le fichier sont imbriqués et chaque élément json s'étend sur plusieurs lignes du fichier.

employees4.json

```
[ {
    "nom_complet":{
        "prenom":"Pascal",
        "surnom":"",
        "nom":"Martin"
    },
    "date_naiss":"1995-02-25",
    "genre":"M",
    "salaire":4500
},
{
    "nom_complet":{
        "prenom":"Pierre",
        "surnom":"Rose",
        "nom":""
    },
    "date_naiss":"1998-07-12",
    "genre":"M",
    "salaire":3000
},
{
    "nom_complet":{
```

L'exemple ci-dessous montre comment créer un dataframe à partir de ce fichier.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
```

```
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

df =
spark.read.option("multiline","true").json("hdfs://nnde01/user/mk/json_data
/input/employees4.json")
df.printSchema()
df.show(truncate=False)
```

```
# Output printSchema() df
| -- date_naiss: string(nullable=true)
| -- genre: string(nullable=true)
| -- nom_complet: struct(nullable=true)
| | -- nom: string(nullable=true)
| | -- prenom: string(nullable=true)
| | -- surnom: string(nullable=true)
| -- salaire: long(nullable=true)

# Output show() df
+----------+-----+----------------------------+-------+
|date_naiss|genre|nom_complet                 |salaire|
+----------+-----+----------------------------+-------+
|1995-02-25|M    |[Martin, Pascal, ]          |4500   |
|1998-07-12|M    |[, Pierre, Rose]            |3000   |
|2000-09-14|M    |[Bernard, Laurent, ]        |3000   |
|1989-10-06|F    |[Carpentier, Virginie, Anne]|2500   |
|1982-02-15|F    |[Bouvier, Jenny, Carole]    |3500   |
+----------+-----+----------------------------+-------+
```

4.2.1.4 Créer le dataframe à partir d'un fichier parquet : la fonction read.parquet()

L'exemple ci-dessous montre l'utilisation de la fonction read.parquet() pour lire un fichier parquet stocké sur hdfs.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal', '', 'Martin', '1995-02-25', 'M', 4500),
        ('Pierre', 'Rose', '', '1998-07-12', 'M', 3000),
        ('Laurent', '', 'Bernard', '2000-09-14', 'M', 3000),
        ('Virginie', 'Anne', 'Carpentier', '1989-10-06', 'F', 2500),
        ('Jenny', 'Carole', 'Bouvier', '1982-02-15', 'F', 3500)]
```

```
columns = ["prenom", "surnom", "nom", "date_naiss", "genre", "salaire"]
df = spark.createDataFrame(data=data, schema=columns)
# Ecrire en format pour relecture
df.write.parquet("hdfs://nnde01/user/mk/parquet_data/input/data.parquet")

# Lecture du fichier parquet
df1 =
spark.read.parquet("hdfs://nnde01/user/mk/parquet_data/input/data.parquet")
df1.printSchema()
df1.show()
```

Dans cet exemple, nous créons d'abord un dataframe à partir d'une collection de données. On stocke d'abord ce dataframe sur hdfs sous format parquet afin de pouvoir créer un dataframe à partir de ce parquet avec read.parquet().

```
# Output printSchema() df
| -- prenom: string(nullable=true)
| -- surnom: string(nullable=true)
| -- nom: string(nullable=true)
| -- date_naiss: string(nullable=true)
| -- genre: string(nullable=true)
| -- salaire: long(nullable=true)

# Output show() df
+--------+------+----------+----------+-----+-------+
|  prenom|surnom|       nom|date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Virginie|  Anne|Carpentier|1989-10-06|    F|   2500|
|   Jenny|Carole|   Bouvier|1982-02-15|    F|   3500|
| Laurent|      |   Bernard|2000-09-14|    M|   3000|
|  Pascal|      |    Martin|1995-02-25|    M|   4500|
|  Pierre|  Rose|          |1998-07-12|    M|   3000|
+--------+------+----------+----------+-----+-------+
```

Tout comme la fonction SaveAsTextFile() pour le RDD, la fonction write.parquet() pour le Dataframe renvoie une erreur lorsque le répertoire de stockage existe déjà.

4.3 Créer un dataframe à partir d'une table hive : la fonction spark.sql()

Il est possible de créer un dataframe à partir d'une table hive. Pour cela, il suffit d'appeler le module spark.sql() avec comme argument une requête select sur la table hive source. L'exemple ci-dessous illustre la création d'un dataframe à partir d'une table hive.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
```

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

# Création du dataframe à partir de la table tb_movies
df=spark.sql("select movieid,title,genres from mydatabase.tb_movies order
by movieid")
df.show(n=10,truncate=False)
df.printSchema()
```

```
# Output show() df
+-------+--------------------+--------------------+
|movieid|               title|              genres|
+-------+--------------------+--------------------+
|      1|    Toy Story (1995)|Adventure|Animati...|
|      6|         Heat (1995)|Action|Crime|Thri...|
|     15|Cutthroat Island ...|Action|Adventure|...|
|     20|  Money Train (1995)|Action|Comedy|Cri...|
|     21|   Get Shorty (1995)|Comedy|Crime|Thri...|
|     22|      Copycat (1995)|Crime|Drama|Horro...|
|     29|City of Lost Chil...|Adventure|Drama|F...|
|     32|Twelve Monkeys (a...|Mystery|Sci-Fi|Th...|
|     33|Wings of Courage ...|Adventure|Romance...|
|     47|Seven (a.k.a. Se7...|    Mystery|Thriller|
+-------+--------------------+--------------------+
only showing top 10 rows
# output printSchema df
root
 |-- movieid: integer (nullable = true)
 |-- title: string (nullable = true)
 |-- genres: string (nullable = true)
```

4.4 Créer un dataframe à partir d'un topic Kafka : la fonction read.format("kafka")

La fonction read.format("kafka") permet de créer un dataframe en lisant le contenu d'un topic kafka. Le bout de code ci-dessous permet de lire une file kafka et construire un dataframe à partir des messages lus.

Dans le code ci-dessous, nous préparons d'abord quelques fonctions utiles pour lire gérer les offsets des messages kafka et commiter ces offsets sur hdfs. La gestion des offsets permet de créer le dataframe en sélectionnant les messages qui n'ont pas déjà été prédemment lus. Voir les détails du code ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
```

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json
import subprocess
import os

##########################################################################
##########
############################## Définition des fonctions utilitaires
##################
##########################################################################
##########

"""Fonction pour executer une commande hdfs avec python"""
def run_shell_cmd(hdfs_cmd):
    try:
        proc = subprocess.Popen( hdfs_cmd, shell=True
,stdout=subprocess.PIPE, stderr=subprocess.PIPE)
        cmd_output, cmd_err = proc.communicate()
        cmd_return =  proc.returncode
        cmd_output = cmd_output.decode('ascii')
        cmd_err = cmd_err.decode('ascii')
    except Exception as e :
        sys.exit(1)
    return( cmd_output, cmd_err , cmd_return)

"""Fonction pour tester si un répertoire hdfs existe ou pas"""
def hdfs_dir_exists(hdfs_path):
    res = False
    hdfs_cmd = 'hdfs dfs -test -d {0} ;echo $? '.format(hdfs_path)
    cmd_output, cmd_err , cmd_return = run_shell_cmd(hdfs_cmd)
    if(str(cmd_output)[0] == '0'):
        res= True
        print(' Le DIR : {0} Existe sur HDFS '.format(hdfs_path))
    else:
        print(' Le DIR : {0} n existe pas sur HDFS '.format(hdfs_path))
    return(res)

"""Fonction pour créer un répertoire hdfs avec python """
def create_directory_in_hdfs(hdfs_dir_path):
    hdfs cmd = 'hdfs dfs -mkdir -p {0} '.format(hdfs dir path)
    cmd_output, cmd_err , cmd_return =run_shell_cmd(hdfs_cmd)
    if(cmd_return != 0):
        print("ERROR" in cmd_err)
        sys.exit(1)
    return(True)

"""Fonction pour lire un fichier stocké sur hdfs"""
def read_hdfs_file(hdfs_file_path):
    hdfs_cmd = 'hdfs dfs -cat {0}'.format(hdfs_file_path)
    cmd_output, cmd_err , cmd_return = run_shell_cmd(hdfs_cmd)
    if(cmd_return != 0):
        raise FileNotFoundError("file not found :
{}".format(hdfs_file_path), hdfs_file_path)
    return(cmd_output)

""" Fonction pour copier un fichier du local vers hdfs """
def copyFromLocal_to_hdfs(local_file, hdfs_dir):
```

```
    hdfs_cmd = 'hdfs dfs -copyFromLocal -f  {0}
{1}'.format(local_file,hdfs_dir)
    cmd_output, cmd_err , cmd_return = run_shell_cmd(hdfs_cmd)
    if(cmd_return != 0):
        print(' Erreur lors du upload sur HDFS : {0}'.format(cmd_err))
        sys.exit(1)
""" Fonction pour récupérer les offsets kafka stockés sur hdfs """
def retrieve_offsets(checkpoint_folder):
    try:
        if not hdfs_dir_exists(checkpoint_folder):
            create_directory_in_hdfs(checkpoint_folder)
        offset_file_content =
read_hdfs_file(os.path.join(checkpoint_folder, offset_filename))
        offsets = json.loads(offset_file_content)
    except FileNotFoundError: # le fichier d offset n existe pas si c est
la 1ere execution
        offsets = None
    except: # erreur dans le parsing du contenu du fichier
        raise IOError()
    return offsets
""" Fonction pour commiter les offsets kafka sur hdfs"""
def commit_offsets(offsets_to_commit,checkpoint_folder):
    if offsets_to_commit != None:
        if not hdfs_dir_exists(checkpoint_folder):
            create_directory_in_hdfs(checkpoint_folder)
        with open(offset_filename, "w") as file:
            file.write(json.dumps(offsets_to_commit))
        copyFromLocal_to_hdfs(offset_filename, checkpoint_folder)

""" Fonction pour consumer une file kafka et préparer les offsets à
commiter """
def consume_kafka_topic(topic,bootstrap_server,checkpoint_folder)  :
    global offsets_to_commit
    offsets = retrieve_offsets(checkpoint_folder)
    if offsets is not None:
        offset_dict = {}
        offset_dict[topic] = offsets
        offsets = json.dumps(offset_dict)
    else:
        offsets = "earliest"
    df =spark.read \
        .format("kafka") \
        .option("kafka.bootstrap.servers", bootstrap_server) \
        .option("kafka.security.protocol", "SASL_PLAINTEXT") \
        .option("subscribe", topic) \
        .option("startingOffsets", offsets) \
        .load()
    # Caster la value du message en json string
    df=df.withColumn("value",F.col("value")\
            .cast(StringType())).cache()
    # Récup des derniers offet de chaque parition kafka en vue du commit
des offsets
    offsetRow = df.select(F.col('partition'), F.col('offset')) \
            .groupBy('partition') \
            .agg(F.max(F.col("offset")).alias("offset_fin")) \
            .collect()
    if (len(offsetRow) > 0):
        offsets_to_commit = {str(row["partition"]): int(row["offset_fin"])
+ 1 for row in offsetRow} # # incrementation de 1 offset de fin pour ne pas
le reconsommer lors de la prochaine execution
    return df
```

```
##########################################################################
##########
############################## Lancement du job kafka
################################
##########################################################################
##########

""" Définition des variables de lancement """
spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
bootstrap_server="mdvyy31001.cdr.bigdata.esd.fr:6667,mdvyy31002.cdr.bigdata
.esd.fr:6667"
topic="mykafkaTopic"
checkpoint_folder="hdfs://nnde01/user/mk/tmp/kafka_test/checkpointLocation"
offsets_to_commit=None
offset_filename="offset_{}".format(topic)

## Consommer la file kafka
df=consume_kafka_topic(topic, bootstrap_server, checkpoint_folder)
cols_to_select=df.columns
cols_to_select.remove("key")
df=df.select(*cols_to_select)
df.select("value").show(truncate=False)
df.printSchema()
#Commiter les offsets à la fin du traitement des messages.
commit_offsets(offsets_to_commit,checkpoint_folder)
```

```
# Output show() df["value"]
+-----------------------------------------------+
|value                                          |
+-----------------------------------------------+
|{"nom":"Sylvain","ville":"Paris","age":22}     |
|{"nom":"Laurent","ville":"New York","age":25}  |
|{"nom":"Alice","ville":"Londres","age":27}     |
+-----------------------------------------------+

# Output printSchema df
root
 |-- value: string (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)
```

4.5 Examiner la structure et le contenu d'un Dataframe : les fonctions printSchema() et show() et l'attribut columns

La fonction printSchema() affiche le schéma d'un dataframe c'est à dire les colonnes et leur type. La fonction show() permet d'afficher un nombre défini de lignes du dataframe. L'attribut columns renvoie la liste des colonnes dans un dataframe. Les examples ci-dessous montrent l'utilisation de chacune de ces fonctions et attribut.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)

print(df.columns)
df.printSchema()
df.show(n=5,truncate=False)
```

```
# Output df.columns
['prenom', 'surnom', 'nom', 'date_naiss', 'genre', 'salaire']

# Output printSchema() df
| -- prenom: string(nullable=true)
| -- surnom: string(nullable=true)
| -- nom: string(nullable=true)
| -- date_naiss: string(nullable=true)
| -- genre: string(nullable=true)
| -- salaire: long(nullable=true)


# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+
```

4.6 Transformations d'un Dataframe

4.6.1 Ajouter une nouvelle colonne à un dataframe ou modifier une colonne existante dans un dataframe : la fonction withColumn()

La fonction withColumn() permet de créer une nouvelle colonne ou modifier une colonne déjà existante dans un dataframe.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()


data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]


columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

#Ajouter une colonne donnant le nom complet
df1= df.withColumn("nom_complet", F.concat(F.col("prenom"),F.lit(" "),
F.col("surnom"), F.lit(" "),F.col("nom")))
#Modifier une colonne existante
df1= df1.withColumn("salaire",F.col("salaire")*1.10) #multiplier chaque
élément par 10%
df1.show(truncate=False)
```

```
# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+
```

```
# Output show() df1
+--------+------+----------+----------+-----+----------+-----------------------+
|prenom  |surnom|nom       |date_naiss|genre|salaire   |nom_complet            |
+--------+------+----------+----------+-----+----------+-----------------------+
|Pascal  |      |Martin    |1995-02-25|M    |4950.0    |Pascal  Martin         |
|Pierre  |Rose  |          |1998-07-12|M    |3300.0    |Pierre Rose            |
|Laurent |      |Bernard   |2000-09-14|M    |3300.0    |Laurent  Bernard       |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2750.0    |Virginie Anne Carpentier|
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3850.0    |Jenny Carole Bouvier   |
+--------+------+----------+----------+-----+----------+-----------------------+
```

4.6.2 Sélectionner une ou plusieurs colonnes dans un Dataframe : la fonction select()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()


data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]


columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

# Sélectionner la colonne prenom
# méthode 1
df1=df.select("prenom")
# méthode 2
df1=df.select(df["prenom"])
# méthode 3
df1=df.select(F.col("prenom"))

df1.show(truncate=False)

# sélectionner plusieurs colonnes
cols_to_select=["prenom","surnom","nom"]
df2=df.select(*cols_to_select)
df2.show(truncate=False)

#Sélectionner les colonnes selon leur index dans la liste des colonnes
df3=df.select(df.columns[:3]) # les trois premières colonnes
```

```
df3.show(truncate=False)
df4=df.select(df.columns[-3 :]) # les trois dernières colonnes
df4.show(truncate=False)
df5=df.select(df.columns[2 :5]) # selectionner trois colonnes entre l'index
2 et 5
df5.show(truncate=False)
```

```
# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df1
+--------+
|prenom  |
+--------+
|Pascal  |
|Pierre  |
|Laurent |
|Virginie|
|Jenny   |
+--------+

# Output show() df2
+--------+------+----------+
|prenom  |surnom|nom       |
+--------+------+----------+
|Pascal  |      |Martin    |
|Pierre  |Rose  |          |
|Laurent |      |Bernard   |
|Virginie|Anne  |Carpentier|
|Jenny   |Carole|Bouvier   |
+--------+------+----------+

# Output show() df3
+--------+------+----------+
|prenom  |surnom|nom       |
+--------+------+----------+
|Pascal  |      |Martin    |
|Pierre  |Rose  |          |
|Laurent |      |Bernard   |
|Virginie|Anne  |Carpentier|
|Jenny   |Carole|Bouvier   |
+--------+------+----------+

# Output show() df4
+----------+-----+-------+
|date_naiss|genre|salaire|
+----------+-----+-------+
|1995-02-25|M    |4500   |
```

```
|1998-07-12|M     |3000    |
|2000-09-14|M     |3000    |
|1989-10-06|F     |2500    |
|1982-02-15|F     |3500    |
+----------+------+--------+

# Output show() df5
+----------+----------+------+
|nom       |date_naiss|genre|
+----------+----------+------+
|Martin    |1995-02-25|M    |
|          |1998-07-12|M    |
|Bernard   |2000-09-14|M    |
|Carpentier|1989-10-06|F    |
|Bouvier   |1982-02-15|F    |
+----------+----------+------+
```

4.6.3 Tester si une colonne existe dans un Dataframe : l'attribut columns

L'attribut columns appliqué à un dataframe renvoie tous les noms des colonnes du dataframe sous forme de liste. Pour vérifier si une colonne existe dans un dataframe, il suffit vérifier que la colonne dans la liste renvoyée par l'attribut. Voir exemple ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.printSchema()
df.show(truncate=False)

# Tester si la colonne "prenom" se trouve dans le dataframe df
bool1=True if "prenom" in [ col_name.lower() for col_name in df.columns]
else False
print(bool1) # renvoie True

# Tester si la colonne "id" se trouve dans le dataframe
bool2=True if "id" in [ col_name.lower() for col_name in df.columns] else
False
```

```
print(bool2) # renvoie False
```

4.6.4 Renommer les colonnes dans un dataframe : la fonction withColumnRenamed() ou les fonctions select()+alias()

La façon la plus naturelle pour renommer une colonne dans un Dataframe est d'utiliser la fonction withColumRenamed(). Mais on peut aussi utiliser la fonction alias() pour renommer une colonne. Cependant à la différence de la fonction withColumnRenamed(), la fonction alias() doit être utilisé avec la fonction select(). En effet, lors de la sélection des colonnes d'un dataframe, spark offre la possibilité de les renommer directement en utilisant des alias. Ces alias deviennent les nouveaux noms dans le dataframe de sortie. Les exemples ci-dessous montrent comment renommer une colonne en utilisant les deux méthodes.

4.6.4.1 Cas 1 : Renommer une colonne bien définie : la fonction withColumnRenamed()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()


data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]


columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

# renommer la colonne date_naiss en date_of_birth avec withColumnRenamed()
df1=df.withColumnRenamed("date_naiss","date_of_birth")
df1.show(truncate=False)
```

```
# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df1
+--------+------+----------+-------------+-----+-------+
|prenom  |surnom|nom       |date_of_birth|genre|salaire|
+--------+------+----------+-------------+-----+-------+
|Pascal  |      |Martin    |1995-02-25   |M    |4500   |
|Pierre  |Rose  |          |1998-07-12   |M    |3000   |
|Laurent |      |Bernard   |2000-09-14   |M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06   |F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15   |F    |3500   |
+--------+------+----------+-------------+-----+-------+
```

4.6.4.2Cas 2 : Sélectionner et renommer plusieurs colonnes : la fonction select()+alias()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()


data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]


columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

# Sélection et renommer chaque colonne du dataframe
df1=df.select(
              F.col("prenom").alias("first_name")
              ,F.col("surnom").alias("nickname")
              ,F.col("nom").alias("family_name")
              ,F.col("date_naiss").alias("date_of_birth")
```

```
               ,F.col("genre")
               ,F.col("salaire")
                 )

df1.show(truncate=False)
```

```
# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df1
+----------+--------+-----------+-------------+------+------+
|first_name|nickname|family_name|date_of_birth|genre|salaire|
+----------+--------+-----------+-------------+------+------+
|Pascal    |        |Martin     |1995-02-25   |M     |4500  |
|Pierre    |Rose    |           |1998-07-12   |M     |3000  |
|Laurent   |        |Bernard    |2000-09-14   |M     |3000  |
|Virginie  |Anne    |Carpentier |1989-10-06   |F     |2500  |
|Jenny     |Carole  |Bouvier    |1982-02-15   |F     |3500  |
+----------+--------+-----------+-------------+------+------+
```

4.6.4.3Cas 3 : Renommer toutes les colonnes dans un dataframe

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()


data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
```

```
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]


columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)


# Renommer toutes les colonnes suivant une liste définie
old_columns=df.columns
new_columns=["first_name","nickname","family_name","date_of_birth","gender"
,"salary"]
aliases=[]
for old, new in zip(old_columns, new_columns) :
        aliases.append(F.col(old).alias(new))
df1 = df.select(*aliases)
df1.show(truncate=False)
```

```
# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+
+----------+--------+-----------+-------------+------+------+
|first_name|nickname|family_name|date_of_birth|gender|salary|
+----------+--------+-----------+-------------+------+------+
|Pascal    |        |Martin     |1995-02-25   |M     |4500  |
|Pierre    |Rose    |           |1998-07-12   |M     |3000  |
|Laurent   |        |Bernard    |2000-09-14   |M     |3000  |
|Virginie  |Anne    |Carpentier |1989-10-06   |F     |2500  |
|Jenny     |Carole  |Bouvier    |1982-02-15   |F     |3500  |
+----------+--------+-----------+-------------+------+------+
```

4.6.5 Ajouter un préfixe/suffixe au nom de toutes les colonnes d'un dataframe : la fonction select()+alias()

Faisons d'abord remarquer qu'ajouter un préfixe ou un suffixe au nom d'une colonne constitue de facto une opération de renommage de cette colonne. Ainsi, ajouter un préfixe ou un suffixe au nom de l'ensemble des colonnes revient à renommer toutes les colonnes du dataframe. Nous avons déjà montré cette opération dans la section « Renommer les colonnes d'un dataframe ». Dans la présente section, il s'agit d'un cas

spécifique dans lequel la liste des colonnes de sortie est construite en partant des colonnes déjà présentes dans le dataframe. Voir illustration dans les exemples ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json


spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()


data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]


columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)


# Cas 1: Ajouter un préfixe au nom de l'ensemble des colonnes
prefix="emp_"
new_cols=[]
for old_name in df.columns:
        new_name=f"{prefix}{old_name}"
        new_cols.append(F.col(old_name).alias(new_name))
df1 = df.select(*new_cols)
df1.show(truncate=False)

# Cas 2: Ajouter un suffixe au nom de l'ensemble des colonnes
suffix="_emp"
new_cols=[]
for old_name in df.columns:
        new_name=f"{old_name}{suffix}"
        new_cols.append(F.col(old_name).alias(new_name))
df2= df.select(*new_cols)
df2.show(truncate=False)
```

```
# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
```

```
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df1
+----------+----------+----------+--------------+---------+-----------+
|emp_prenom|emp_surnom|emp_nom   |emp_date_naiss|emp_genre|emp_salaire|
+----------+----------+----------+--------------+---------+-----------+
|Pascal    |          |Martin    |1995-02-25    |M        |4500       |
|Pierre    |Rose      |          |1998-07-12    |M        |3000       |
|Laurent   |          |Bernard   |2000-09-14    |M        |3000       |
|Virginie  |Anne      |Carpentier|1989-10-06    |F        |2500       |
|Jenny     |Carole    |Bouvier   |1982-02-15    |F        |3500       |
+----------+----------+----------+--------------+---------+-----------+

# Output show() df2
+----------+----------+----------+--------------+---------+-----------+
|prenom_emp|surnom_emp|nom_emp   |date_naiss_emp|genre_emp|salaire_emp|
+----------+----------+----------+--------------+---------+-----------+
|Pascal    |          |Martin    |1995-02-25    |M        |4500       |
|Pierre    |Rose      |          |1998-07-12    |M        |3000       |
|Laurent   |          |Bernard   |2000-09-14    |M        |3000       |
|Virginie  |Anne      |Carpentier|1989-10-06    |F        |2500       |
|Jenny     |Carole    |Bouvier   |1982-02-15    |F        |3500       |
+----------+----------+----------+--------------+---------+-----------+
```

4.6.6 Supprimer un préfixe/suffixe du nom de toutes les colonnes d'un dataframe

Tout comme pour l'ajout de préfixe et de suffixe, faisons encore remarquer que supprimer un préfixe ou un suffixe du nom d'une colonne constitue une opération de renommage de cette colonne. Ainsi, supprimer un préfixe ou un suffixe du nom de l'ensemble des colonnes revient à renommer toutes les colonnes du dataframe. Nous avons déjà montré cette opération dans la section « Renommer les colonnes d'un dataframe ». Dans la présente section, il s'agit d'un cas spécifique dans lequel la liste des colonnes de sortie est construite en partant des colonnes déjà présentes dans le dataframe. Voir illustration dans les exemples ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()


# Cas 1: Supprimer le préfixe "emp_" du nom de l'ensemble des colonnes
```

```
data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns =
["emp_prenom","emp_surnom","emp_nom","emp_date_naiss","emp_genre","emp_sala
ire"]
prefixed_df = spark.createDataFrame(data=data, schema = columns)
prefixed_df.show(truncate=False)

# Supprimer le prefix
prefix="emp_"
new_cols=[]
for old_name in prefixed_df.columns:
        new_name=old_name[len(prefix):] if
old_name.lower().startswith(prefix.lower()) else old_name
        new_cols.append(F.col(old_name).alias(new_name))
df1 = prefixed_df.select(*new_cols)
df1.show(truncate=False)

# Cas 2: Supprimer le suffixe "_emp" du nom de l'ensemble des colonnes

data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns =
["prenom_emp","surnom_emp","nom_emp","date_naiss_emp","genre_emp","salaire_
emp"]
suffixed_df = spark.createDataFrame(data=data, schema = columns)
suffixed_df.show(truncate=False)
# Supprimer le suffix
suffix="_emp"
new_cols=[]
for old_name in suffixed_df.columns:
        new_name=old_name[:-len(suffix)] if
old_name.lower().endswith(suffix.lower()) else old_name
        new_cols.append(F.col(old_name).alias(new_name))
df2=suffixed df.select(*new cols)
df2.show(truncate=False)
```

```
# Output show() prefixed_df
+----------+----------+----------+--------------+---------+-----------+
|emp_prenom|emp_surnom|emp_nom   |emp_date_naiss|emp_genre|emp_salaire|
+----------+----------+----------+--------------+---------+-----------+
|Pascal    |          |Martin    |1995-02-25    |M        |4500       |
|Pierre    |Rose      |          |1998-07-12    |M        |3000       |
|Laurent   |          |Bernard   |2000-09-14    |M        |3000       |
|Virginie  |Anne      |Carpentier|1989-10-06    |F        |2500       |
|Jenny     |Carole    |Bouvier   |1982-02-15    |F        |3500       |
+----------+----------+----------+--------------+---------+-----------+
```

```
# Output show() df1
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+


# Output show() suffixed_df
+----------+----------+----------+--------------+---------+-----------+
|prenom_emp|surnom_emp|nom_emp   |date_naiss_emp|genre_emp|salaire_emp|
+----------+----------+----------+--------------+---------+-----------+
|Pascal    |          |Martin    |1995-02-25    |M        |4500       |
|Pierre    |Rose      |          |1998-07-12    |M        |3000       |
|Laurent   |          |Bernard   |2000-09-14    |M        |3000       |
|Virginie  |Anne      |Carpentier|1989-10-06    |F        |2500       |
|Jenny     |Carole    |Bouvier   |1982-02-15    |F        |3500       |
+----------+----------+----------+--------------+---------+-----------+

# Output show() df1
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+
```

4.6.7 Récupérer le schéma d'un dataframe (les colonnes et leur type) : les attributs dtypes et schema.fields

Deux méthodes sont utilisées ici pour récupérer le schema d'un dataframe : l'attribut dtypes ou l'attribut schema.fields. L'attribut dtypes appliqué à un dataframe permet de renvoyer les colonnes et leur type sous forme d'une liste de tuple (colonne, type) où le type est indiqué sous son format string. L'attribut schema.field renvoie une liste d'objet colonne ayant deux propriétés à savoir name qui renvoie le nom de la colonne et dataType qui renvoie le type de la colonne sous son format classe Spark. Les exemples ci-dessus illustrent l'utilisation des deux méthodes pour récupérer le schéma d'un dataframe.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json
```

```
spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [(1,"M",23,150.0,"2022-06-10","2022-06-10 10:25:15.13"),
        (2,"M",21,180.0,"2022-06-10","2022-06-10 10:26:12.03"),
        (3,"F",20,161.0,"2022-06-10","2022-06-10 10:27:26.13"),
        (4,"M",23,168.0,"2022-06-10","2022-06-10 10:45:11.45"),
        (5,"F",31,170.0,"2022-06-10","2022-06-10 10:52:26.17"),
        (6,"F",22,175.0,"2022-06-10","2022-06-10 11:08:03.03")]

columns=["id_coureur","genre","age","taille","date_course","heure_arrivee"]
df = spark.createDataFrame(data=data, schema = columns)
# positionner les types des colonnes
df=df.select(F.col("id_coureur").cast("long")
            ,F.col("genre").cast("string")
            ,F.col("age").cast("integer")
            ,F.col("taille").cast("double")
            ,F.col("date_course").cast("date")
            ,F.col("heure_arrivee").cast("timestamp")
            )
df.printSchema()
df.show(truncate=False)

# Méthode 1: Récuperer chaque colonne et son type (string) l'attribut
dtypes
schema_string=df.dtypes
print(schema_string)
# Méthode 2: Récupérer chaque colonne et son type (classe) l'attribut
schema.fields
schema_class=[(col.name,col.dataType) for col in df.schema.fields]
print(schema_class)
```

```
#output printSchema() df

root
 |-- id_coureur: long (nullable = true)
 |-- genre: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- taille: double (nullable = true)
 |-- date_course: date (nullable = true)
 |-- heure_arrivee: timestamp (nullable = true)

# Output show() df
+----------+-----+---+------+-----------+---------------------+
|id_coureur|genre|age|taille|date_course|heure_arrivee        |
+----------+-----+---+------+-----------+---------------------+
|1         |M    |23 |150.0 |2022-06-10 |2022-06-10 10:25:15.13|
|2         |M    |21 |180.0 |2022-06-10 |2022-06-10 10:26:12.03|
|3         |F    |20 |161.0 |2022-06-10 |2022-06-10 10:27:26.13|
|4         |M    |23 |168.0 |2022-06-10 |2022-06-10 10:45:11.45|
|5         |F    |31 |170.0 |2022-06-10 |2022-06-10 10:52:26.17|
|6         |F    |22 |175.0 |2022-06-10 |2022-06-10 11:08:03.03|
+----------+-----+---+------+-----------+---------------------+

#output print schema_string
```

```
[('id_coureur', 'bigint'), ('genre', 'string'), ('age', 'int'), ('taille',
'double'), ('date_course', 'date'), ('heure_arrivee', 'timestamp')]
#output print schema_class
[('id_coureur', LongType), ('genre', StringType), ('age', IntegerType),
('taille', DoubleType), ('date_course', DateType), ('heure_arrivee',
TimestampType)]
```

Note importante : Remarquons ici que lorsqu'une colonne est castée en type "long", le type de cette colonne sera toujours du "bigint" sous format string lorsqu'on récupérera le schéma. En revanche, son type sous le format classe restera toujours LongType. De même, lorsqu'on caste la colonne en type "bigint", le format "bigint" sera certes renvoyé lorsqu'on récupère le schéma avec dtypes mais le type sera LongType lorsqu'on récupère le schéma avec schema.fields. Il est important de faire cette précision, car il permet d'éviter de faire des erreurs de spécification lorsqu'on veut par exemple faire des tests sur les types des colonnes. Par exemples, tester si une colonne est de type string, integer, double, long, etc. C'est pourquoi, lorsqu'il s'agira de faire des tests sur les types des colonnes, il est préférable de récupérer le schéma de dataframe en utilisant l'attribut schema.fields. On pourra toujours utiliser l'attribut dtypes à conditions de bien s'assurer que le type format string que l'on indique dans le test soit correct. Par exemples, pour un test sur le type long, il faut spécifier "bigint". Et pour un test sur le type integer, il faut spécifier "int".

Le tableau ci-dessous fournit les correspondances entre les types en format string et les types en format classe.

Types spark en format string(renvoyés par dtypes)	Types en format classe (renvoyés par schema.fields.dataType
"string"	StringType
"integer" ou "int"	IntegerType
"double"	DoubleType
"bigint" ou "long"	LongType
"boolean"	BooleanType
"date"	DateType
"timestamp"	TimestampType

Les principaux types Spark sont disponibles à ce lien

https://spark.apache.org/docs/latest/sql-ref-datatypes.html

4.6.8 Tester le type d'une colonne présente dans un dataframe: l'attribut dtypes ou la fonctions isinstance().

Il existe différente façons de tester le type d'une colonne dans un dataframe. On peut soit utiliser l'attribut dtypes ou utiliser la fonction isinstance(). Cependant l'utilisation de ces fonctions pour tester le type de la colonne n'est pas directe. Il faut d'abord passer par quelques transformations. Les exemples ci-dessous permettent de tester le type d'une colonne présente dans un dataframe.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [(1,"M",23,150.0,"2022-06-10","2022-06-10 10:25:15.13"),
        (2,"M",21,180.0,"2022-06-10","2022-06-10 10:26:12.03"),
        (3,"F",20,161.0,"2022-06-10","2022-06-10 10:27:26.13"),
        (4,"M",23,168.0,"2022-06-10","2022-06-10 10:45:11.45"),
        (5,"F",31,170.0,"2022-06-10","2022-06-10 10:52:26.17"),
        (6,"F",22,175.0,"2022-06-10","2022-06-10 11:08:03.03")]

columns=["id_coureur","genre","age","taille","date_course","heure_arrivee"]
#Create du dataframe
df = spark.createDataFrame(data=data, schema = columns)
df=df.select(F.col("id_coureur").cast("integer")
            ,F.col("genre").cast("string")
            ,F.col("age").cast("integer")
            ,F.col("taille").cast("double")
            ,F.col("date_course").cast("date")
            ,F.col("heure_arrivee").cast("timestamp")
            )
df.printSchema()
df.show(truncate=False)

#### Tester que la colonne "id_coureur" est de type integer (int)

# Méthode 1: utiliser la fonction dtypes
if dict(df.dtypes)["id_coureur"].lower()=="int":
    print("id_coureur","int",True)
else:
    print("id_coureur","int",False)

# Méthode 2: utiliser la fonction isinstance()
for field in df.schema.fields:
    if field.name.lower()=="id_coureur" and isinstance(field.dataType,
IntegerType):
        print("id_coureur",field.dataType,True)
    elif field.name.lower()=="id_coureur" and not
isinstance(field.dataType, IntegerType):
        print("id_coureur",field.dataType,False)

####### Tester que la colonne "genre" est de type string

if dict(df.dtypes)["genre"].lower()=="string":
```

```
    print("genre","string",True)
else:
    print("genre","string",False)

# Méthode 2: utiliser la fonction isinstance()
for field in df.schema.fields:
    if field.name.lower()=="genre" and isinstance(field.dataType,
StringType):
        print("genre",field.dataType,True)
    elif field.name.lower()=="genre" and not isinstance(field.dataType,
StringType):
        print("genre",field.dataType,False)
```

```
#output printSchema() df

root
 |-- id_coureur: integer (nullable = true)
 |-- genre: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- taille: double (nullable = true)
 |-- date_course: date (nullable = true)
 |-- heure_arrivee: timestamp (nullable = true)

# Output show() df

+----------+-----+---+------+-----------+---------------------+
|id_coureur|genre|age|taille|date_course|heure_arrivee        |
+----------+-----+---+------+-----------+---------------------+
|1         |M    |23 |150.0 |2022-06-10 |2022-06-10 10:25:15.13|
|2         |M    |21 |180.0 |2022-06-10 |2022-06-10 10:26:12.03|
|3         |F    |20 |161.0 |2022-06-10 |2022-06-10 10:27:26.13|
|4         |M    |23 |168.0 |2022-06-10 |2022-06-10 10:45:11.45|
|5         |F    |31 |170.0 |2022-06-10 |2022-06-10 10:52:26.17|
|6         |F    |22 |175.0 |2022-06-10 |2022-06-10 11:08:03.03|
+----------+-----+---+------+-----------+---------------------+

# Output Tests type

# Output méthode 1
id_coureur int True
id_coureur IntegerType True

# Output méthode 2
genre string True
genre StringType True
```

Le tableau ci-dessous donne les principaux types des colonnes dans un dataframe Spark.

Types spark à utiliser avec l'attribut dtypes	Types spark à utiliser avec la fonction isinstance()

"string"	StringType
"integer" ou "int"	IntegerType
"double"	DoubleType
"bigint"	LongType
"boolean"	BooleanType
"date"	DateType
"timestamp"	TimestampType

Les types Spark sont disponibles à ce lien

https://spark.apache.org/docs/latest/sql-ref-datatypes.html

4.6.9 Sélectionner toutes les colonnes d'un certain type : l'attribut dtypes et la fonction isinstance()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [(1,"M",23,150.0,"2022-06-10","2022-06-10 10:25:15.13"),
        (2,"M",21,180.0,"2022-06-10","2022-06-10 10:26:12.03"),
        (3,"F",20,161.0,"2022-06-10","2022-06-10 10:27:26.13"),
        (4,"M",23,168.0,"2022-06-10","2022-06-10 10:45:11.45"),
        (5,"F",31,170.0,"2022-06-10","2022-06-10 10:52:26.17"),
        (6,"F",22,175.0,"2022-06-10","2022-06-10 11:08:03.03")]

columns=["id_coureur","genre","age","taille","date_course","heure_arrivee"]
#Create du dataframe
df = spark.createDataFrame(data=data, schema = columns)
df=df.select(F.col("id_coureur").cast("integer")
            ,F.col("genre").cast("string")
            ,F.col("age").cast("integer")
            ,F.col("taille").cast("double")
            ,F.col("date_course").cast("date")
            ,F.col("heure_arrivee").cast("timestamp")
            )
df.printSchema()
df.show(truncate=False)

# Sélectionner toutes les colonnes de type integer

Méthode 1: utiliser la fonction dtypes
integer_cols = [col for col,type in dict(df.dtypes).items() if
type.lower()=="int"]
```

```
df1=df.select(*integer_cols)
df1.show( truncate=False)
Méthode 2: utiliser la fonction isinstance()
integer cols = [field.name for field in df.schema.fields if
isinstance(field.dataType, IntegerType)]
df1=df.select(*integer_cols)
df1.show( truncate=False)

# Sélectionner toutes les colonnes de type string

Méthode 1: utiliser la fonction dtypes
string_cols = [col for col,type in dict(df.dtypes).items() if
type.lower()=="string"]
df2=df.select(*string_cols)
df2.show( truncate=False)

Méthode 2: utiliser la fonction isinstance()
string_cols = [field.name for field in df.schema.fields if
isinstance(field.dataType, StringType)]
df2=df.select(*string_cols)
df2.show( truncate=False)

# Sélectionner toutes les colonnes de date

Méthode 1: utiliser la fonction dtypes
date_cols = [col for col,type in dict(df.dtypes).items() if
type.lower()=="date"]
df3=df.select(*date_cols)
df3.show( truncate=False)
Méthode 2: utiliser la fonction isinstance()
date_cols = [field.name for field in df.schema.fields if
isinstance(field.dataType, DateType)]
df3=df.select(*date_cols)
df3.show( truncate=False)
```

```
#output printSchema() df

root
 |-- id_coureur: integer (nullable = true)
 |-- genre: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- taille: double (nullable = true)
 |-- date_course: date (nullable = true)
 |-- heure_arrivee: timestamp (nullable = true)

# Output show() df

+----------+-----+---+------+-----------+---------------------+
|id_coureur|genre|age|taille|date_course|heure_arrivee        |
+----------+-----+---+------+-----------+---------------------+
|1         |M    |23 |150.0 |2022-06-10 |2022-06-10 10:25:15.13|
|2         |M    |21 |180.0 |2022-06-10 |2022-06-10 10:26:12.03|
|3         |F    |20 |161.0 |2022-06-10 |2022-06-10 10:27:26.13|
|4         |M    |23 |168.0 |2022-06-10 |2022-06-10 10:45:11.45|
|5         |F    |31 |170.0 |2022-06-10 |2022-06-10 10:52:26.17|
```

```
|6          |F    |22 |175.0 |2022-06-10 |2022-06-10 11:08:03.03|
+----------+-----+---+------+-----------+----------------------+

# Output show() df1

+----------+---+
|id_coureur|age|
+----------+---+
|1         |23 |
|2         |21 |
|3         |20 |
|4         |23 |
|5         |31 |
|6         |22 |
+----------+---+

# Output show() df2

+-----+
|genre|
+-----+
|M    |
|M    |
|F    |
|M    |
|F    |
|F    |
+-----+
# Output show() df3

+-----------+
|date_course|
+-----------+
|2022-06-10 |
|2022-06-10 |
|2022-06-10 |
|2022-06-10 |
|2022-06-10 |
|2022-06-10 |
+-----------+
```

Le tableau ci-dessous donne les principaux types des colonnes dans un dataframe Spark.

Types spark à utiliser avec l'attribut dtypes	Types spark à utiliser avec la fonction isinstance()
"string"	StringType
"integer" ou "int"	IntegerType
"double"	DoubleType
"bigint"	LongType
"boolean"	BooleanType
"date"	DateType
"timestamp"	TimestampType

Les types Spark sont disponibles à ce lien

https://spark.apache.org/docs/latest/sql-ref-datatypes.html

4.6.10 Changer le type d'une colonne dans un dataframe cast() et asType()

On peut changer le type d'une colonne en utilisant soit la fonction cast(), soit la fonction asType(). Pour chacune des deux fonctions, on peut spécifier le type de sortie soit sous format string, soit sous format classe. Les exemples ci-dessous montrent différentes méthodes pour changer le type d'une colonne dans un dataframe.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()


data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]


columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)
df.printSchema()

# Caster la colonne date_naiss en type date.
#méthode 1
df1=df.withColumn("date_naiss", F.col("date_naiss").cast("date"))
df1.printSchema()
#méthode 2
df2=df.withColumn("date_naiss", F.col("date_naiss").cast(DateType()))
df2.printSchema()
#méthode 3
df3=df.withColumn("date_naiss", F.col("date_naiss").astype("date"))
df3.printSchema()
#méthode 4
df4=df.withColumn("date_naiss", F.col("date_naiss").astype(DateType()))
df4.printSchema()
```

```
# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+


# Output printSchema() df
 |-- prenom: string (nullable = true)
 |-- surnom: string (nullable = true)
 |-- nom: string (nullable = true)
 |-- date_naiss: string (nullable = true)
 |-- genre: string (nullable = true)
 |-- salaire: long (nullable = true)

# Output printSchema() Méthode 1
 |-- prenom: string (nullable = true)
 |-- surnom: string (nullable = true)
 |-- nom: string (nullable = true)
 |-- date_naiss: date (nullable = true)
 |-- genre: string (nullable = true)
 |-- salaire: long (nullable = true)

# Output printSchema() Méthode 2
| -- prenom: string(nullable=true)
| -- surnom: string(nullable=true)
| -- nom: string(nullable=true)
| -- date_naiss: date(nullable=true)
| -- genre: string(nullable=true)
| -- salaire: long(nullable=true)

# Output printSchema() Méthode 3

| -- prenom: string(nullable=true)
| -- surnom: string(nullable=true)
| -- nom: string(nullable=true)
| -- date naiss: date(nullable=true)
| -- genre: string(nullable=true)
| -- salaire: long(nullable=true)

# Output printSchema() Méthode 4

| -- prenom: string(nullable=true)
| -- surnom: string(nullable=true)
| -- nom: string(nullable=true)
| -- date_naiss: date(nullable=true)
| -- genre: string(nullable=true)
| -- salaire: long(nullable=true)
```

Le tableau ci-dessous donne les principaux types des colonnes dans un dataframe Spark.

Types spark format string	Types spark format classe
"string"	StringType
"integer" ou "int"	IntegerType
"double"	DoubleType
"long" ou "bigint"	LongType
"boolean"	BooleanType
"date"	DateType
"timestamp"	TimestampType

Les types Spark sont disponibles à ce lien

https://spark.apache.org/docs/latest/sql-ref-datatypes.html

4.6.11 Trier un dataframe selon les valeurs d'une colonne : les fonction sort() et orderBy()

Les fonctions sort() et orderBy() permettent de trier les lignes d'un dataframe en se basant sur une ou plusieurs colonnes. Voir exemples ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()


data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]


columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

# Utiliser la fonction sort()
# Trier selon les valeurs croissantes de date_naiss (date de naissance)

df1=df.sort("date_naiss") #tri par ordre croissant
df1=df.sort(F.col("date_naiss").asc()) #tri par ordre croissant
df1=df.sort(F.col("date_naiss").asc_nulls_first()) #tri par ordre
```

```
croissant, les valeurs nulles positionnées en premier
df1=df.sort(F.col("date_naiss").asc_nulls_last()) #tri par ordre croissant,
les valeurs nulles positionnées en dernier

df1.show(truncate=False)
#trier par ordre décroissant
df1=df.sort(F.col("date_naiss").desc())
df1=df.sort(F.col("date_naiss").desc_nulls_first()) #tri par ordre
décroissant, les valeurs nulles positionnées en premier
df1=df.sort(F.col("date_naiss").desc_nulls_last()) #tri par ordre
décroissant, les valeurs nulles positionnées en dernier

# Trier selon les valeurs croissantes de date_naiss (date de naissance) et
genre
df2=df.sort("date_naiss" ,"genre")
df2.show(truncate=False)
#trier par ordre décroissant de date_naiss (date de naissance) et genre
df3=df.sort(F.col("date_naiss").desc() , F.col("genre").desc())
df3.show(truncate=False)

# Trier en utilisant la fonction orderBy()
# Trier selon les valeurs croissantes de date_naiss (date de naissance)

df4=df.orderBy("date_naiss") #tri par ordre croissant
df4=df.orderBy(F.col("date_naiss").asc()) #tri par ordre croissant
df4=df.orderBy(F.col("date_naiss").asc_nulls_first()) #tri par ordre
croissant, les valeurs nulles positionnées en premier
df4=df.orderBy(F.col("date_naiss").asc_nulls_last()) #tri par ordre
croissant, les valeurs nulles positionnées en dernier
df4.show(truncate=False)
#trier par ordre décroissant
df5=df.orderBy(F.col("date_naiss").desc())
df5=df.orderBy(F.col("date_naiss").desc_nulls_first()) #tri par ordre
décroissant, les valeurs nulles positionnées en premier
df5=df.orderBy(F.col("date_naiss").desc_nulls_last()) #tri par ordre
décroissant, les valeurs nulles positionnées en dernier
df5.show(truncate=False)

# Trier selon les valeurs croissantes de date_naiss (date de naissance) et
genre
df6=df.orderBy("date_naiss" ,"genre")
df6.show(truncate=False)
#trier par ordre décroissant de date_naiss (date de naissance) et genre
df7=df.orderBy(F.col("date_naiss").desc() , F.col("genre").desc())
df7.show(truncate=False)
```

```
# Output show() df1
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
+--------+------+----------+----------+-----+-------+
# Output show() df2
+--------+------+----------+----------+-----+-------+
```

```
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
+--------+------+----------+----------+-----+-------+

# Output show() df3
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df4
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
+--------+------+----------+----------+-----+-------+

# Output show() df5
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df6
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
+--------+------+----------+----------+-----+-------+

# Output show() df7
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
```

```
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+
```

4.6.12 Supprimer les doublons dans un dataframe : les fonctions distinct() et dropDuplicates()

La fonction distinct() permet de supprimer les lignes dupliquées dans un dataframe. Les lignes sont dites dupliquées lorsque pour deux ou plusieurs lignes, les valeurs des colonnes sont toutes identiques. La fonction distinct() garde une seule ligne et supprime les autres.

Alors que la fonction distinct() supprime les doublons lorsque les valeurs de toutes les colonnes sont identiques, la fonction dropDuplicates() supprime les doublons en se basant uniquement sur quelques colonnes bien identifiées.

Les exemples ci-dessous illustrent l'utilisation des fonctions distinct() et dropDuplicates()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal','','Martin','1995-02-25','M',4500),
      ('Pascal','','Martin','1995-02-25','M',4500),
      ('Pascal','','Martin','1995-02-25','M',4500),
      ('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

# supprimer les doublons à partir de toutes les colonnes
df1=df.distinct()
df1.show(truncate=False)

#Supprimer les doublons à partir de quelques colonnes sélectionnées
df2=df.dropDuplicates(["prenom","surnom","nom"])
df2.show(truncate=False)
```

```
# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df1
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
+--------+------+----------+----------+-----+-------+

# Output show() df2
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
+--------+------+----------+----------+-----+-------+
```

4.6.13 Utiliser les opérateurs logiques et les fonctions de test usuels dans les fonctions de transformation d'un dataframe

Les opérateurs logiques sont souvent utilisés pour définir une condition permettant soit de filtrer les lignes d'un dataframe, soit créer une nouvelle colonne ou modifier une colonne existante. Le tableau ci-dessous présente les opérateurs logiques usuels d'un dataframe utilisables sur un dataframe.

Symbole opérateur logique	Description
==	Egal à
>	Supérieur à

>=	Supérieur ou égal à
<	Inférieur à
<=	Inférieur ou égal à
!=	Différent de
~	Not : n'est pas
&	Et
|	ou
contains()	Contient (chaîne de caractères)
startswith()	Commence par (chaîne de caractères)
endswith()	Se termine par (chaîne de caractères)
like()	Permet de généraliser les fonctions contains(), startswith() et endsWith()
isin()	Se trouve dans (une liste de valeurs)
isNull()	Est nul
isNotNull()	N'est pas nul
isnan()	N'est pas un nombre (Not a Number)

A noter que les opérateurs logiques sont souvent utilisés lors de l'appel d'une fonction filter() ou d'une fonction where(). Ils sont aussi utilisés lors de l'appel de la fonction withColumn() lorsque cet appel sert à créer une colonne conditionnelle. Une colonne conditionnelle est une colonne dont les valeurs sont définies à partir d'une ou de plusieurs conditions appliquée(s) sur d'autres colonnes. Ces conditions sont alors définies avec les fonctions when()/otherwise(). Nous reviendrons plus tard sur l'utilisation de la fonction withColumn() en combinaison avec les conditions when()/otherwise().

Les exemples présentées ci-dessous montrent l'utilisation des opérateurs logiques et fonctions de tests usuels dans une spécification filter().

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]
```

```
columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.printSchema()
df.show(truncate=False)

df.persist()

# Utiliser l'opérateur & (et)
#Filtrer les lignes dont genre est "F" et salaire >3000
df1=df.filter((F.col("genre")=="F") & (F.col("salaire")>3000)) #
df1.show(truncate=False)

# Utiliser l'opérateur | (ou)
#Filtrer les lignes dont genre est "F" ou salaire >3000
df2=df.filter((F.col("genre")=="F") | (F.col("salaire")>3000)) #
df2.show(truncate=False)

# Utiliser l'opérateur ~ (not)
#Filtrer les lignes dont genre n'est pas "F"
df3=df.filter( ~( F.col("genre")=="F"))
df3.show(truncate=False)

# Utiliser la fonction contains()
#Filtrer les lignes dont prenom contient "P"
df4=df.filter( F.col("prenom").contains("P")) #
df4.show(truncate=False)
# Utiliser la fonction startsWith()
#Filtrer les lignes dont prenom commence par "P"
df5=df.filter( F.col("prenom").startswith("P")) #
df5.show(truncate=False)

# Utiliser la fonction endsWith()
#Filtrer les lignes dont nom se termine par "r"
df6=df.filter(F.col("nom").endswith("r")) #
df6.show(truncate=False)

# Utiliser la fonction like()
#Filtrer les lignes dont prenom contient 'P'
df7=df.filter( F.col("prenom").like('%P%')) #
df7.show(truncate=False)
#Filtrer les lignes dont prenom commence par 'P'
df8=df.filter( F.col("prenom").like('P%')) #
df8.show(truncate=False)
#Filtrer les lignes dont nom finit par 'r'
df9=df.filter( F.col("nom").like('%r')) #
df9.show(truncate=False)

# Utiliser la fonction isNotNull()
#Filtrer les lignes dont surnom n'est pas nul
df10=df.filter( F.col("surnom").isNotNull()) #
df10.show(truncate=False)

# Utiliser la fonction isin()
#Filtrer les lignes dont prenom est dans la liste 'Pascal', 'Pierre',
'Virginie'
df11=df.filter( F.col("prenom").isin('Pascal', 'Pierre', 'Virginie')) #
df11.show(truncate=False)

df.unpersist()
```

```
# Output printSchema() df
 |-- prenom: string (nullable = true)
 |-- surnom: string (nullable = true)
 |-- nom: string (nullable = true)
 |-- date_naiss: string (nullable = true)
 |-- genre: string (nullable = true)
 |-- salaire: long (nullable = true)

# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df1
+------+------+-------+----------+-----+-------+
|prenom|surnom|nom    |date_naiss|genre|salaire|
+------+------+-------+----------+-----+-------+
|Jenny |Carole|Bouvier|1982-02-15|F    |3500   |
+------+------+-------+----------+-----+-------+

# Output show() df2
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df3
+-------+------+-------+----------+-----+-------+
|prenom |surnom|nom    |date_naiss|genre|salaire|
+-------+------+-------+----------+-----+-------+
|Pascal |      |Martin |1995-02-25|M    |4500   |
|Pierre |Rose  |       |1998-07-12|M    |3000   |
|Laurent|      |Bernard|2000-09-14|M    |3000   |
+-------+------+-------+----------+-----+-------+

# Output show() df4
+------+------+------+----------+-----+-------+
|prenom|surnom|nom   |date_naiss|genre|salaire|
+------+------+------+----------+-----+-------+
|Pascal|      |Martin|1995-02-25|M    |4500   |
|Pierre|Rose  |      |1998-07-12|M    |3000   |
+------+------+------+----------+-----+-------+

# Output show() df5
+------+------+------+----------+-----+-------+
|prenom|surnom|nom   |date_naiss|genre|salaire|
+------+------+------+----------+-----+-------+
|Pascal|      |Martin|1995-02-25|M    |4500   |
|Pierre|Rose  |      |1998-07-12|M    |3000   |
+------+------+------+----------+-----+-------+
```

```
# Output show() df6
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df7
+------+------+------+----------+-----+-------+
|prenom|surnom|nom   |date_naiss|genre|salaire|
+------+------+------+----------+-----+-------+
|Pascal|      |Martin|1995-02-25|M    |4500   |
|Pierre|Rose  |      |1998-07-12|M    |3000   |
+------+------+------+----------+-----+-------+
# Output show() df8
+------+------+------+----------+-----+-------+
|prenom|surnom|nom   |date_naiss|genre|salaire|
+------+------+------+----------+-----+-------+
|Pascal|      |Martin|1995-02-25|M    |4500   |
|Pierre|Rose  |      |1998-07-12|M    |3000   |
+------+------+------+----------+-----+-------+

# Output show() df9
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df10
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df11
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
+--------+------+----------+----------+-----+-------+
```

4.6.14 Appliquer un filtre sur les lignes d'un dataframe: les fonctions filter() ou where()

Pour filtrer les lignes d'un dataframe suivant une condition bien définie, on peut utiliser soit la fonction filter() ou la fonction where(). Ces deux fonctions sont complètement interchangeables. Les exemples ci-dessous montrent l'utilisation de chacune des deux fonctions.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)


######### Filtrer les lignes dont genre est "F" et salaire >3000
df1_a=df.filter((F.col("genre")=="F") & (F.col("salaire")>3000)) #
df1_a.show(truncate=False)
df1_b=df.where((F.col("genre")=="F") & (F.col("salaire")>3000)) #
df1_b.show(truncate=False)


######### Filtrer les lignes dont genre est "F" ou salaire >3000
df2_a=df.filter((F.col("genre")=="F") | (F.col("salaire")>3000)) #
df2_a.show(truncate=False)
df2_b=df.where((F.col("genre")=="F") | (F.col("salaire")>3000)) #
df2_b.show(truncate=False)


######## Filtrer les lignes dont genre n'est pas "F"
df3_a=df.filter( ~( F.col("genre")=="F"))
df3_a.show(truncate=False)
df3_b=df.where( ~( F.col("genre")=="F"))
df3_b.show(truncate=False)
```

```
# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
```

```
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+
# Output show() df1_a
+------+------+-------+----------+-----+-------+
|prenom|surnom|nom    |date_naiss|genre|salaire|
+------+------+-------+----------+-----+-------+
|Jenny |Carole|Bouvier|1982-02-15|F    |3500   |
+------+------+-------+----------+-----+-------+

# Output show() df1_b
+------+------+-------+----------+-----+-------+
|prenom|surnom|nom    |date_naiss|genre|salaire|
+------+------+-------+----------+-----+-------+
|Jenny |Carole|Bouvier|1982-02-15|F    |3500   |
+------+------+-------+----------+-----+-------+

# Output show() df2_a
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df2_b
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df3_a
+-------+------+-------+----------+-----+-------+
|prenom |surnom|nom    |date_naiss|genre|salaire|
+-------+------+-------+----------+-----+-------+
|Pascal |      |Martin |1995-02-25|M    |4500   |
|Pierre |Rose  |       |1998-07-12|M    |3000   |
|Laurent|      |Bernard|2000-09-14|M    |3000   |
+-------+------+-------+----------+-----+-------+

# Output show() df3_b
+-------+------+-------+----------+-----+-------+
|prenom |surnom|nom    |date_naiss|genre|salaire|
+-------+------+-------+----------+-----+-------+
|Pascal |      |Martin |1995-02-25|M    |4500   |
|Pierre |Rose  |       |1998-07-12|M    |3000   |
|Laurent|      |Bernard|2000-09-14|M    |3000   |
+-------+------+-------+----------+-----+-------+
```

4.6.15 Tester si la valeur d'une colonne contient une chaîne de caractère donnée : les fonctions contains() et like()

```
# Import de toutes les librairies utilitaires usuelles
```

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)
# Filter toutes les lignes pour lesquelles le nom contient le caractère
"ti".
# utiliser contains()
df1=df.filter(F.lower(F.col("nom")).contains("ti")) # méthode 1
df1.show(truncate=False)
# utiliser like()
df1=df.filter(F.lower(F.col("nom")).like("%ti%")) # méthode 2
df1.show(truncate=False)
```

```
# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df1 avec contains()
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
+--------+------+----------+----------+-----+-------+

# Output show() df1 avec like()
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
+--------+------+----------+----------+-----+-------+
```

4.6.16 Tester si la valeur d'une colonne commence par une chaîne de caractère donnée : la fonction startswith()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

# Filter toutes les lignes pour lesquelles le prenom commence par le
caractère "P".
df1=df.filter(F.lower(F.col("prenom")).startswith("p"))
df1.show()
```

```
# Output show() df
+----------+------+----------+----------+-----+-------+
|prenom    |surnom|nom       |date_naiss|genre|salaire|
+----------+------+----------+----------+-----+-------+
|Pascal    |      |Martin    |1995-02-25|M    |4500   |
|Pierre    |Rose  |          |1998-07-12|M    |3000   |
|Laurent   |      |Bernard   |2000-09-14|M    |3000   |
|Virginie  |Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny     |Carole|Bouvier   |1982-02-15|F    |3500   |
+----------+------+----------+----------+-----+-------+

# Output show() df1
+---------+------+------+----------+-----+-------+
|   prenom|surnom|   nom|date_naiss|genre|salaire|
+---------+------+------+----------+-----+-------+
|Pascal   |      |Martin|1995-02-25|    M|   4500|
|   Pierre|  Rose|      |1998-07-12|    M|   3000|
+---------+------+------+----------+-----+-------+
```

4.6.17 Tester si la valeur d'une colonne se termine par une chaîne de caractère donnée : la fonction endswith()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

# Filter toutes les lignes pour lesquelles le nom se termine par le
caractère "er".
df1=df.filter(F.lower(F.col("nom")).endswith("er"))
df1.show()
```

```
# Output show() df
+----------+------+----------+----------+-----+-------+
|prenom    |surnom|nom       |date_naiss|genre|salaire|
+----------+------+----------+----------+-----+-------+
|Pascal    |      |Martin    |1995-02-25|M    |4500   |
|Pierre    |Rose  |          |1998-07-12|M    |3000   |
|Laurent   |      |Bernard   |2000-09-14|M    |3000   |
|Virginie  |Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny     |Carole|Bouvier   |1982-02-15|F    |3500   |
+----------+------+----------+----------+-----+-------+

# Output show() df1
+--------+------+----------+----------+-----+-------+
|  prenom|surnom|       nom|date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Virginie|  Anne|Carpentier|1989-10-06|    F|   2500|
|   Jenny|Carole|   Bouvier|1982-02-15|    F|   3500|
+--------+------+----------+----------+-----+-------+
```

4.6.18 Tester si la valeur d'une colonne est nulle ou non nulle : les fonctions isNull(), isNotNull()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal',None,'Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie',None,'Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

# Filter toutes les lignes où surnom est null.
df1=df.filter(F.col("surnom").isNull())
df1.show(truncate=False)
# Filter toutes les lignes où surnom est non null.
df2=df.filter(F.col("surnom").isNotNull())
df2.show(truncate=False)
```

```
# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |null  |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|null  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df1
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |null  |Martin    |1995-02-25|M    |4500   |
|Virginie|null  |Carpentier|1989-10-06|F    |2500   |
+--------+------+----------+----------+-----+-------+

# Output show() df2
+-------+------+-------+----------+-----+-------+
|prenom |surnom|nom    |date_naiss|genre|salaire|
```

```
+-------+------+-------+----------+-----+-------+
|Pierre |Rose  |       |1998-07-12|M    |3000   |
|Laurent|      |Bernard|2000-09-14|M    |3000   |
|Jenny  |Carole|Bouvier|1982-02-15|F    |3500   |
+-------+------+-------+----------+-----+-------+
```

4.6.19 Tester si la valeur d'une colonne se trouve dans une liste de valeurs : la fonction isin()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal',None,'Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie',None,'Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

# Filtrer toutes les lignes où nom se trouve dans la liste : 'Martin',
'Bernard', 'Bouvier'
df1=df.filter(F.col("nom").isin('Martin', 'Bernard', 'Bouvier'))
df1.show(truncate=False)
```

```
# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |null  |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|null  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df1
+-------+------+-------+----------+-----+-------+
|prenom |surnom|nom    |date_naiss|genre|salaire|
+-------+------+-------+----------+-----+-------+
|Pascal |null  |Martin |1995-02-25|M    |4500   |
|Laurent|      |Bernard|2000-09-14|M    |3000   |
```

```
|Jenny   |Carole|Bouvier|1982-02-15|F     |3500    |
+--------+------+-------+----------+------+--------+
```

4.6.20 Tester si la valeur d'une colonne contient une expression régulière : la fonction rlike()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal_35','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie_03','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

# Filter toutes les lignes pour lesquelles le prenom ne contient que des
lettres alphabétiques.
df1=df.filter(F.lower(F.col("prenom")).rlike("^[a-z]*$"))
df1.show(truncate=False)

# Filter toutes les lignes pour lesquelles le prenom contient des chiffres.
df2=df.filter(F.col("prenom").rlike(".*[0-9].*"))
df2.show(truncate=False)
```

```
# Output show() df
+-----------+------+----------+----------+-----+-------+
|prenom     |surnom|nom       |date_naiss|genre|salaire|
+-----------+------+----------+----------+-----+-------+
|Pascal_35  |      |Martin    |1995-02-25|M    |4500   |
|Pierre     |Rose  |          |1998-07-12|M    |3000   |
|Laurent    |      |Bernard   |2000-09-14|M    |3000   |
|Virginie_03|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny      |Carole|Bouvier   |1982-02-15|F    |3500   |
+-----------+------+----------+----------+-----+-------+

# Output show() df1
+-------+------+-------+----------+-----+-------+
|prenom |surnom|nom    |date_naiss|genre|salaire|
+-------+------+-------+----------+-----+-------+
```

```
|Pierre |Rose  |       |1998-07-12|M    |3000   |
|Laurent|      |Bernard|2000-09-14|M    |3000   |
|Jenny  |Carole|Bouvier|1982-02-15|F    |3500   |
+-------+------+-------+----------+-----+-------+

# Output show() df2
+-----------+------+----------+----------+-----+-------+
|prenom     |surnom|nom       |date_naiss|genre|salaire|
+-----------+------+----------+----------+-----+-------+
|Pascal_35  |      |Martin    |1995-02-25|M    |4500   |
|Virginie_03|Anne  |Carpentier|1989-10-06|F    |2500   |
+-----------+------+----------+----------+-----+-------+
```

4.6.21 Extraire et renvoyer une chaîne de caractère de la valeur d'une colonne : fonction substring()

La fonction substring() permet d'extraire et de renvoyer une chaine de caractère à partir de la valeur d'une colonne. Les exemples ci-dessous montrent l'utilisation de la fonction substring().

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()


data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]


columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

# Extraire l'année de naissance à partir de la date de naissance date_naiss
df1=df.withColumn("annee_naiss", F.substring(F.col("date_naiss"),0,4)) #
extrait l'année et crée une nouvelle colonne nommée annee_naiss (year of
birth)
df1.show(truncate=False)
```

```
# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df1
+--------+------+----------+----------+-----+-------+-----------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|annee_naiss|
+--------+------+----------+----------+-----+-------+-----------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |1995       |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |1998       |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |2000       |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |1989       |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |1982       |
+--------+------+----------+----------+-----+-------+-----------+
```

4.6.22 Concatener plusieurs colonnes en une seule colonne : les fonction concat() et concat_ws()

La fonction concat() permet de concatener la valeur de plusieurs colonnes sans aucun séparateur. La fonction concat_ws() permet de concatener les valeurs de plusieurs colonnes en laissant la possibilité d'indiquer un séparateur de type string. Voir exemples ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]
```

```
columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)

df.show(truncate=False)

cols_list=["prenom","surnom","nom"]
# Concatener les colonnes "prenom","surnom","nom" sans séparateur
df1=df.withColumn("name_s", F.concat(*cols_list) )
df1.show(truncate=False)
# Concatener les colonnes "prenom","surnom","nom" avec séparateur
df2=df.withColumn("full_name", F.concat_ws(" ",*cols_list) ) # séparateur
espace
df2.show(truncate=False)
```

```
# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+
# Output show() df1
+--------+------+----------+----------+-----+-------+----------------------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|name_s                |
+--------+------+----------+----------+-----+-------+----------------------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |PascalMartin          |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |PierreRose            |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |LaurentBernard        |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |VirginieAnneCarpentier|
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |JennyCaroleBouvier    |
+--------+------+----------+----------+-----+-------+----------------------+
# Output show() df2
+--------+------+----------+----------+-----+-------+------------------------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|full_name               |
+--------+------+----------+----------+-----+-------+------------------------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |Pascal  Martin          |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |Pierre Rose             |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |Laurent  Bernard        |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |Virginie Anne Carpentier|
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |Jenny Carole Bouvier    |
+--------+------+----------+----------+-----+-------+------------------------+
```

4.6.23 Splitter la valeur d'une colonne et générer plusieurs colonnes : la fonction split() et getItem()

L'exemple ci-dessous montre comment splitter la valeur d'une colonne et générer plusieurs colonnes à partir de la valeur splittée. Le split de la valeur d'une colonne se fait en indiquant un séparateur, ex : ",", " ;", " ", etc.

Lorsque le split est réalisé sur la valeur de la colonne, le résultat renvoyée est une séquence de valeurs accessible par leur index : 0,1,2, etc. La fonction qui permet de

récupérer la valeur associée à un index est getItem(). Dans l'exemple ci-dessous, la colonne movie_ref est splittée pour générer trois colonnes supplémentaires :movieId, title et genres.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [("1;Toy Story
(1995);Adventure|Animation|Children|Comedy|Fantasy",),
            ("2;Jumanji (1995);Adventure|Children|Fantasy",),
            ("3;Grumpier Old Men (1995);Comedy|Romance",),
            ("4;Waiting to Exhale (1995);Comedy|Drama|Romance",),
            ("5;Father of the Bride Part II (1995);Comedy",),
  ]

columns = ["movie_ref"]
df1 = spark.createDataFrame(data=data, schema = columns)
df1.printSchema()
df1.show(truncate=False)

# Construire le dataframe splité
df2=df1.select(F.split(F.col("movie_ref"),";").getItem(0).alias("movieId")
               ,F.split(F.col("movie_ref"),";").getItem(1).alias("title")
               ,F.split(F.col("movie_ref"),";").getItem(2).alias("genres"))

df2.printSchema()
df2.show(truncate=False)
```

```
# Output printSchema() df1
|-- movie ref: string (nullable = true)

# Output show() df1
+------------------------------------------------------------+
|movie_ref                                                   |
+------------------------------------------------------------+
|1;Toy Story (1995);Adventure|Animation|Children|Comedy|Fantasy|
|2;Jumanji (1995);Adventure|Children|Fantasy                  |
|3;Grumpier Old Men (1995);Comedy|Romance                    |
|4;Waiting to Exhale (1995);Comedy|Drama|Romance              |
|5;Father of the Bride Part II (1995);Comedy                 |
+------------------------------------------------------------+

# Output printSchema() df2
| -- movieId: string(nullable=true)
| -- title: string(nullable=true)
| -- genres: string(nullable=true)

# Output show() df2
+-------+---------------------------------+-----------------------------------------+
```

```
|movieId|title                              |genres                                       |
+-------+-----------------------------------+---------------------------------------------+
|1      |Toy Story (1995)                   |Adventure|Animation|Children|Comedy|Fantasy|
|2      |Jumanji (1995)                     |Adventure|Children|Fantasy                   |
|3      |Grumpier Old Men (1995)            |Comedy|Romance                               |
|4      |Waiting to Exhale (1995)           |Comedy|Drama|Romance                         |
|5      |Father of the Bride Part II (1995)|Comedy                                       |
+-------+-----------------------------------+---------------------------------------------+..
```

4.6.24 Rechercher et remplacer une chaîne de caractères par une autre chaîne de caractères dans la valeur d'une colonne : la fonction regexp_replace()

La fonction regexp_replace() permet de remplacer une chaîne de caractères (généralement un pattern) par une autre dans la valeur d'une colonne. L'exemple ci-dessous montre l'utilisation de la fonction regexp_replace()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal','','Martin_xxx','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard_xxx','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier_xxx','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier_xxx','1982-02-15','F',3500)]

columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]

df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)
# Remplacer _xxx par "" dans la colonne nom
# regexp_replace
df1=df.withColumn("nom", F.regexp_replace("nom","_xxx","") )
df1.show(truncate=False)
```

```
# Output show() df
+--------+------+--------------+----------+-----+-------+
|prenom  |surnom|nom           |date_naiss|genre|salaire|
+--------+------+--------------+----------+-----+-------+
|Pascal  |      |Martin_xxx    |1995-02-25|M    |4500   |
|Pierre  |Rose  |              |1998-07-12|M    |3000   |
|Laurent |      |Bernard_xxx   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier_xxx|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier_xxx   |1982-02-15|F    |3500   |
+--------+------+--------------+----------+-----+-------+
```

```
# Output show() df1
+--------+------+-----------+----------+-----+-------+
|prenom  |surnom|nom        |date_naiss|genre|salaire|
+--------+------+-----------+----------+-----+-------+
|Pascal  |      |Martin     |1995-02-25|M    |4500   |
|Pierre  |Rose  |           |1998-07-12|M    |3000   |
|Laurent |      |Bernard    |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier |1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier    |1982-02-15|F    |3500   |
+--------+------+-----------+----------+-----+-------+
```

4.6.25 Créer une colonne suivant une ou plusieurs conditions : les fonctions when()/otherwhise()

Les fonction when()/otherwhise() sont des opérateurs conditionnels qui permettent d'appliquer une ou plusieurs conditions sur les colonnes d'un dataframe et de renvoyer un résultat de calcul. Ces fonctions jouent le même rôle que le clause CASE WHEN d'une requête standard (sql, hql, spark.sql). Les fonctions when()/otherwhise() sont souvent utilisées pour ajouter une nouvelle colonne au dataframe ou de modifier une colonne existante. Les colonnes ajoutées/modifiées suite à l'utilisation des fonctions when()/otherwhise() sont appelées colonnes à valeurs conditionnelles. L'exemple ci-dessous illustre un cas d'utilisation des fonctions when()/otherwhise(). Dans l'exemple présenté, il s'agit d'ajouter une colonne nommée sexe qui prend la valeur "Homme", lorsque la colonne genre est égale à "M", qui prend la valeur "Femme", lorsque la colonne genre est égale à "F" et qui prend la valeur NULL dans tout autre cas. Voir code ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal','','Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

# Créer la colonne sexe à partir de la colonne genre
df1=df.withColumn("sexe", F.when(F.col("genre")=="M","Homme")\
```

```
                        .when(F.col("genre")=="F","Femme")
                              .otherwise(None) )
df1.show(truncate=False)
```

```
# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+
# Output show() df1
+--------+------+----------+----------+-----+-------+-----+
|prenom  |surnom|nom       |date_naiss|genre|salaire|sexe |
+--------+------+----------+----------+-----+-------+-----+
|Pascal  |      |Martin    |1995-02-25|M    |4500   |Homme|
|Pierre  |Rose  |          |1998-07-12|M    |3000   |Homme|
|Laurent |      |Bernard   |2000-09-14|M    |3000   |Homme|
|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |Femme|
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |Femme|
+--------+------+----------+----------+-----+-------+-----+
```

4.6.26 Supprimer les colonnes dans un dataframe : la fonction drop()

La fonction drop() permet de supprimer une ou plusieurs colonnes à la fois dans un dataframe. L'exemple ci-dessous montre quelques cas d'utilisation de la fonction drop().

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal',None,'Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie',None,'Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]
```

```
columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

# cas1: supprimer une colonne unique : ex nom
df1=df.drop("nom")
df1.show(truncate=False)
# cas2: supprimer plusieur colonnes : ex surnom, nom
df2=df.drop("surnom","nom")
df2.show(truncate=False)
# cas3 supprimer une liste de colonnnes : ex  ["surnom", "nom","genre"]
cols_list=["surnom", "nom", "genre"]
df3=df.drop(*cols_list)
df3.show(truncate=False)
```

```
# Output show() df
+--------+------+----------+----------+-----+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+-----+-------+
|Pascal  |null  |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|null  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+-----+-------+

# Output show() df1
+--------+------+----------+-----+-------+
|prenom  |surnom|date_naiss|genre|salaire|
+--------+------+----------+-----+-------+
|Pascal  |null  |1995-02-25|M    |4500   |
|Pierre  |Rose  |1998-07-12|M    |3000   |
|Laurent |      |2000-09-14|M    |3000   |
|Virginie|null  |1989-10-06|F    |2500   |
|Jenny   |Carole|1982-02-15|F    |3500   |
+--------+------+----------+-----+-------+

# Output show() df2
+--------+----------+-----+-------+
|prenom  |date_naiss|genre|salaire|
+--------+----------+-----+-------+
|Pascal  |1995-02-25|M    |4500   |
|Pierre  |1998-07-12|M    |3000   |
|Laurent |2000-09-14|M    |3000   |
|Virginie|1989-10-06|F    |2500   |
|Jenny   |1982-02-15|F    |3500   |
+--------+----------+-----+-------+

# Output show() df3
+--------+----------+-------+
|prenom  |date_naiss|salaire|
+--------+----------+-------+
|Pascal  |1995-02-25|4500   |
|Pierre  |1998-07-12|3000   |
|Laurent |2000-09-14|3000   |
|Virginie|1989-10-06|2500   |
```

```
|Jenny   |1982-02-15|3500   |
+--------+----------+-------+
```

4.6.27 Agréger les colonnes d'un dataframe non groupé : count(), mean(), avg(), max(), min() , sum(), etc.

Petit rappel :

Un dataframe non groupé est un dataframe qui ne fait l'objet d'aucun groupage suite à l'application de la fonction groupBy(). Nous reviendrons en détails sur l'utilisation de la fonction groupBy() dans la section suivante « Grouper les données d'un Dataframe » Faire des agrégations sur les colonnes d'un dataframe, c'est calculer les statistiques standards en appliquant des fonction telles que count(), mean(), avg(), max(), min(), sum(), etc. Ces fonctions font partie de la catégorie des fonctions d'agrégation classiques de Spark. Le tableau ci-dessous fournit les détails sur les fonctions d'agrégation les plus couramment utilisées.

A noter ici que lorsqu'on applique une fonction de statistique de base sur un dataframe non groupé avec la fonction groupBy(), le résultat renvoyé est toujours un dataframe constitué d'une seule ligne. Mais lorsqu'il s'agit d'un dataframe avec l'utilisation de la fonction groupBy(), le résultat est un nouveau dataframe qui contient autant de lignes que de valeurs distinctes dans la clé de groupage. Nous reviendrons plus tard sur l'utilisation de la fonction groupBy() dans la section « Grouper les données d'un Dataframe ». Dans l'exemple qui suit, il s'agit de calculer les statistiques de base sur la colonne salaire issue d'un dataframe non groupé.

Les fonction d'agrégation standards	
fonction	**description**
count()	Compte le nombre de valeurs trouvées
countDistinct()	Compte le nombre de valeur distinctes trouvées
avg()	Calcule la moyenne
mean()	Calcule la moyenne
first() ou first(col, ignorenulls=False)	Renvoie la première valeur trouvée
last() ou last(col, ignorenulls=False)	Renvoie la dernière valeur trouvée
max()	Renvoie la valeur max
min()	Renvoie la valeur min
sum()	Renvoie la somme

sumDistinct()	Renvoie la somme des valeurs distinctes
collect_list()	Renvoie toutes les valeurs trouvées sous forme de listes
collect_set()	Renvoie toutes les valeurs sous forme de set (valeurs distinctes)

L'exemple ci-dessous montre l'utilisation des statistiques de base sur les colonnes d'un Dataframe non groupé.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal',None,'Martin','1995-02-25','M',4500),
  ('Pierre','Rose','','1998-07-12','M',3000),
  ('Laurent','','Bernard','2000-09-14','M',3000),
  ('Virginie',None,'Carpentier','1989-10-06','F',2500),
  ('Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

# Calculer les stats sur la colonne salaire: count,mean/avg, sum, min, max
df1=df.select(F.count(F.col("salaire")).alias("count_salaire")
              ,F.mean(F.col("salaire")).alias("mean_salaire")
              ,F.avg(F.col("salaire")).alias("avg_salaire")
              ,F.sum(F.col("salaire")).alias("sum_salaire")
              ,F.min(F.col("salaire")).alias("min_salaire")
              ,F.max(F.col("salaire")).alias("max_salaire")
              )
df1.show(truncate=False)
```

```
# Output show() df
+--------+------+----------+----------+------+-------+
|prenom  |surnom|nom       |date_naiss|genre|salaire|
+--------+------+----------+----------+------+-------+
|Pascal  |null  |Martin    |1995-02-25|M    |4500   |
|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|Virginie|null  |Carpentier|1989-10-06|F    |2500   |
|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+--------+------+----------+----------+------+-------+

# Output show() df1
+-------------+------------+-----------+-----------+-----------+-----------+
|count_salaire|mean_salaire|avg_salaire|sum_salaire|min_salaire|max_salaire|
+-------------+------------+-----------+-----------+-----------+-----------+
```

```
|5            |3300.0      |3300.0      |16500       |2500        |4500        |
+-------------+------------+------------+------------+------------+------------+
```

4.6.28 Grouper les données d'un Dataframe : la fonction groupBy()

La fonction groupBy() permet de grouper les données d'un dataframe suivant un critère défini à partir des valeurs d'une ou de plusieurs colonnes de groupage (clés de groupage). La fonction groupBy() est souvent utilisée dans le but d'appliquer une fonction d'agrégation sur chaque groupe de données et de renvoyer une valeur agrégée pour chaque groupe. Dans un dataframe groupé, les groupes sont définies suivant les valeurs distinctes d'une colonne de groupage ou de la combinaison distinctes des valeurs des colonnes de groupage (clé de groupage). L'exemple ci-dessous montre l'utilisation de la fonction groupBy(). L'exemple montre également l'utilisation des fonctions d'agrégation standards suite au groupage du dataframe à savoir : count(), mean(), avg(), max() , min() et sum().

Rappelons que précédemment, dans la section « Faire des agrégations sur les colonnes d'un dataframe non groupé », nous avons déjà montré que lorsque les fonctions d'agrégation sont appliquées sur un dataframe non groupé, le résultat renvoyé est un nouveau dataframe ayant une seule ligne. Mais lorsque ces mêmes fonctions sont appliquées sur un dataframe groupé suite à l'utilisation de la fonction groupBy(), le résultat renvoyé est alors un dataframe dont le nombre lignes correspond au nombre de valeurs distinctes issues de la clé de groupage.

Dans l'exemple qui suit, nous calculons les statistiques standards sur la colonne salaire dans différents cas de groupage : 1-groupage suivant une seule colonne et 2-groupage suivant plusieurs colonnes. Voir exemple ci-dessous.

Remarque importante : Notons qu'il n'est pas possible de calculer en même temps plusieurs statistiques standards directement sur dataframe groupé. On est obligé de calculer les statistiques l'une après l'autre comme illustré dans l'exemple. Mais lorsqu'on souhaite calculer toutes les statistiques dans une seule spécification, dans ce cas, il faudrait utiliser la fonction agg(). La fonction agg() est une fonction d'agrégation avancée utilisable sur un dataframe groupé et qui permet de calculer plusieurs statistiques standards dans une seule spécification contrairement au cas présent. Pour l'utilisation de la fonction agg(), voir la section suivante « Faire des agrégations sur les colonnes d'un dataframe groupé ».

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
```

```
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal',None,'Martin','1995','M',4500),
  ('Pierre','Rose','','1998','M',3000),
  ('Laurent','','Bernard','1995','M',3000),
  ('Virginie',None,'Carpentier','1998','F',2500),
  ('Jenny','Carole','Bouvier','1998','F',3500)]


columns = ["prenom","surnom","nom","annee_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

#### grouper le dataframe suivant une colonne. ex: genre
df1=df.groupBy("genre")
# Calcul des statistiques
df1.count().show()
df1.mean("salaire").show()
df1.avg("salaire").show()
df1.sum("salaire").show()
df1.min("salaire").show()
df1.max("salaire").show()

# grouper le dataframe suivante plusieurs colonnes: genre, anne_naiss
df2=df.groupBy("genre","annee_naiss")
# Calcul des statistiques
df2.count().show()
df2.mean("salaire").show()
df2.avg("salaire").show()
df2.sum("salaire").show()
df2.min("salaire").show()
df2.max("salaire").show()
```

```
# Output show() df
+--------+------+----------+-----------+-----+-------+
|prenom  |surnom|nom       |annee_naiss|genre|salaire|
+--------+------+----------+-----------+-----+-------+
|Pascal  |null  |Martin    |1995       |M    |4500   |
|Pierre  |Rose  |          |1998       |M    |3000   |
|Laurent |      |Bernard   |1995       |M    |3000   |
|Virginie|null  |Carpentier|1998       |F    |2500   |
|Jenny   |Carole|Bouvier   |1998       |F    |3500   |
+--------+------+----------+-----------+-----+-------+

# Output show() df1
+-----+-----+
|genre|count|
+-----+-----+
|    F|    2|
|    M|    3|
+-----+-----+

+-----+------------+
|genre|avg(salaire)|
+-----+------------+
```

```
|    F|      3000.0|
|    M|      3500.0|
+-----+------------+

+-----+------------+
|genre|avg(salaire)|
+-----+------------+
|    F|      3000.0|
|    M|      3500.0|
+-----+------------+

+-----+------------+
|genre|sum(salaire)|
+-----+------------+
|    F|        6000|
|    M|       10500|
+-----+------------+

+-----+------------+
|genre|min(salaire)|
+-----+------------+
|    F|        2500|
|    M|        3000|
+-----+------------+

+-----+------------+
|genre|max(salaire)|
+-----+------------+
|    F|        3500|
|    M|        4500|
+-----+------------+

# Output show() df2
+-----+-----------+-----+
|genre|annee_naiss|count|
+-----+-----------+-----+
|    F|       1998|    2|
|    M|       1998|    1|
|    M|       1995|    2|
+-----+-----------+-----+

+-----+-----------+------------+
|genre|annee_naiss|avg(salaire)|
+-----+-----------+------------+
|    F|       1998|      3000.0|
|    M|       1998|      3000.0|
|    M|       1995|      3750.0|
+-----+-----------+------------+

+-----+-----------+------------+
|genre|annee_naiss|avg(salaire)|
+-----+-----------+------------+
|    F|       1998|      3000.0|
|    M|       1998|      3000.0|
|    M|       1995|      3750.0|
+-----+-----------+------------+

+-----+-----------+------------+
|genre|annee_naiss|sum(salaire)|
+-----+-----------+------------+
|    F|       1998|        6000|
```

```
|    M|       1998|        3000|
|    M|       1995|        7500|
+-----+-----------+------------+

+-----+-----------+------------+
|genre|annee_naiss|min(salaire)|
+-----+-----------+------------+
|    F|       1998|        2500|
|    M|       1998|        3000|
|    M|       1995|        3000|
+-----+-----------+------------+

+-----+-----------+------------+
|genre|annee_naiss|max(salaire)|
+-----+-----------+------------+
|    F|       1998|        3500|
|    M|       1998|        3000|
|    M|       1995|        4500|
+-----+-----------+------------+
```

A noter que pour calculer les statistiques standard sur une colonne dans un dataframe groupé, on ne peut plus utiliser la fonction select() et appliquer la fonction calcul. On doit directement appliquer la fonction de calcul sur le dataframe. Voir les exemples ci-dessus. Mais l'inconvénient d'appliquer directement une fonction de calcul sur le df c'est qu'on ne peut pas appliquer plusieurs fonctions de calcul dans la même spécification. On est obligé de calculer individuellement chaque statistique. Ce qui n'est pas forcément optimisé.

Mais pour pouvoir calculer ensemble plusieurs statistiques sur le même dataframe groupé, il faut utiliser la fonction agg() qui est la fonction d'agrégation conçue pour pouvoir appliquer plusieurs opérations d'agrégation sur un dataframe groupé. Voir la section suivante.

4.6.29 Agréger les colonnes d'un dataframe groupé : la fonction agg()

La fonction agg() est une fonction d'agrégation avancée applicable sur un dataframe préalablement groupé avec la fonction groupBy() et qui permet d'appliquer les agrégations standards sur les colonnes d'un dataframe : count(), mean(), avg(), max(), min(), sum(), first().

Dans la section précédente « Grouper les données d'un dataframe », nous avons déjà montré qu'il n'est pas possible de calculer directement plusieurs statistiques dans la même spécification sur un dataframe groupé. On a été amené à calculer les statistiques les unes à la suite des autres des autres. Ce qui n'est pas toujours pratique. La fonction agg() constitue la solution face à cette limitation. En effet, en utilisant la fonction agg(), il est possible de spécifier le calcul de plusieurs statistiques en même temps et de renvoyer l'ensemble de ces statistiques dans un seul dataframe. De ce point, la fonction agg() appliquée sur un dataframe groupé joue le même rôle que la fonction select()

appliquée sur un dataframe non groupée. Pour rappel, pour calculer les statistiques standards sur un dataframe non groupé, on utilise la fonction select() à l'intérieur de laquelle on spécifie toutes les fonctions d'agrégation souhaitées (voir la section « Calculer les statistiques standards sur les colonnes d'un dataframe »). En revanche, pour calculer les statistiques standards sur un dataframe groupé, on utilise la fonction agg() à l'intérieur de laquelle on spécifie toutes les fonctions d'agrégation souhaitées (voir exemple d'application ci-dessous)

Le tableau ci-après fournit les fonctions les plus couramment utilisées lors d'une opération d'agrégation.

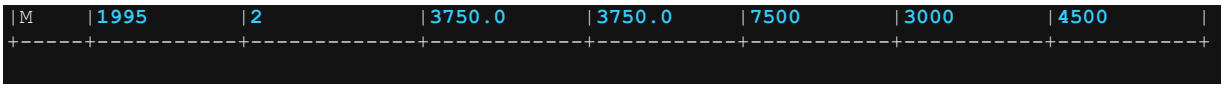
fonction	**description**
count()	Compte le nombre de valeurs trouvées
countDistinct()	Compte le nombre de valeur distinctes trouvées
avg()	Calcule la moyenne
mean()	Calcule la moyenne
first() ou first(col, ignorenulls=False)	Renvoie la première valeur trouvée
last() ou last(col, ignorenulls=False)	Renvoie la dernière valeur trouvée
max()	Renvoie la valeur max
min()	Renvoie la valeur min
sum()	Renvoie la somme
sumDistinct()	Renvoie la somme des valeurs distinctes
collect_list()	Renvoie toutes les valeurs trouvées sous forme de listes
collect_set()	Renvoie toutes les valeurs sous forme de set (valeurs distinctes)

L'exemple ci-dessous illustre l'utilisation de la fonction agg() sur un dataframe

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
```

```
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal',None,'Martin','1995','M',4500),
  ('Pierre','Rose','','1998','M',3000),
  ('Laurent','','Bernard','1995','M',3000),
  ('Virginie',None,'Carpentier','1998','F',2500),
  ('Jenny','Carole','Bouvier','1998','F',3500)]

columns = ["prenom","surnom","nom","annee_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

#### Agréger suivant une colonne. ex: genre
df1=df.groupBy("genre").agg(
              F.count(F.col("salaire")).alias("count_salaire")
              ,F.mean(F.col("salaire")).alias("mean_salaire")
              ,F.avg(F.col("salaire")).alias("avg_salaire")
              ,F.sum(F.col("salaire")).alias("sum_salaire")
              ,F.min(F.col("salaire")).alias("min_salaire")
              ,F.max(F.col("salaire")).alias("max_salaire")
                )

df1.show(truncate=False)

# #### Agréger suivant plusieurs colonnes: genre, anne_naiss
df2=df.groupBy("genre","annee_naiss").agg(
              F.count(F.col("salaire")).alias("count_salaire")
              ,F.mean(F.col("salaire")).alias("mean_salaire")
              ,F.avg(F.col("salaire")).alias("avg_salaire")
              ,F.sum(F.col("salaire")).alias("sum_salaire")
              ,F.min(F.col("salaire")).alias("min_salaire")
              ,F.max(F.col("salaire")).alias("max_salaire")
                )
df2.show(truncate=False)
```

```
# Output show() df
+--------+------+----------+-----------+-----+-------+
|prenom  |surnom|nom       |annee_naiss|genre|salaire|
+--------+------+----------+-----------+-----+-------+
|Pascal  |null  |Martin    |1995       |M    |4500   |
|Pierre  |Rose  |          |1998       |M    |3000   |
|Laurent |      |Bernard   |1995       |M    |3000   |
|Virginie|null  |Carpentier|1998       |F    |2500   |
|Jenny   |Carole|Bouvier   |1998       |F    |3500   |
+--------+------+----------+-----------+-----+-------+

# Output show() df1
+-----+-------------+------------+-----------+-----------+-----------+-----------+
|genre|count_salaire|mean_salaire|avg_salaire|sum_salaire|min_salaire|max_salaire|
+-----+-------------+------------+-----------+-----------+-----------+-----------+
|F    |2            |3000.0      |3000.0     |6000       |2500       |3500       |
|M    |3            |3500.0      |3500.0     |10500      |3000       |4500       |
+-----+-------------+------------+-----------+-----------+-----------+-----------+

# Output show() df2

+-----+-----------+-------------+------------+-----------+-----------+-----------+-----------+
|genre|annee naiss|count salaire|mean salaire|avg salaire|sum salaire|min salaire|max salaire|
+-----+-----------+-------------+------------+-----------+-----------+-----------+-----------+
|F    |1998       |2            |3000.0      |3000.0     |6000       |2500       |3500       |
|M    |1998       |1            |3000.0      |3000.0     |3000       |3000       |3000       |
```



4.6.30 Ajouter une information cumulée ou une information agrégée à un dataframe : la fonction Window.partitionBy() avec ou sans l'option orderBy()

La fonction Window.partitionBy() est très régulièrement sollicitée lors du traitement d'un dataframe à tel point qu'elle apparait parfois incontournable. La spécification de base d'une fonction est Window.partitionBy(). Cependant elle peut être complétée avec l'option orderBy() dans les situations où il s'agit d'appliquer une fonction window standards comme : row_number(), rank(), lag(), lead(), etc.. Ces fonctions windows standards renvoient toujours une information cumulée à l'intérieur de l'unité de partitionnement défini par la fonction partitionBy(). Une information cumulée est une information pour laquelle la valeur courante est obtenue en faisant un incrément (arithmetque ou géométrique) sur la valeur précédente. Ex : le numéro d'ordre, rang, somme cumulée, etc…

Lorsqu'il s'agit de calculer des information agrégées en utilisant les fonctions d'agrégation standards comme : count(), sum(), avg(), mean(), etc l'option orderBy() doit être utilisée avec beaucoup de précaution, sinon au risque de renvoyer des valeurs totalement erronées. Dans les trois sous-sections qui suivent, nous allons présenter quelques cas d'utilisation de la fonction Window().partitionBy() avec ou sans l'option orderBy().

Le tableau ci-dessous montre les fonctions windows (permettant d'obtenir des informations cumulées) et les fonction d'agrégation standards (permettant d'obtenir des informations agrégée) utilisables sur un dataframe.

Fonction	**Description de la fonction**	**Type d'information renvoyée**	**Utilisable avec l'option orderBy() ?**
row_number():	Renvoie une colonne contenant le numéro des lignes par unité de partitionnement	Information cumulée	oui
rank()	Renvoie le rang de chaque ligne par unité de partitionnement. Proche de row_number() à la différence que plusieurs lignes peuvent avoir le même rank si elles ont la même valeur sur le critère de tri.	Information cumulée	oui
lag():	Retourne la valeur de la ligne précédente par unité de partitionnement. Renvoie null pour la première ligne	Information cumulée	oui

Fonction	Description de la fonction	Type d'information renvoyée	Utilisable avec l'option orderBy() ?
lead():	Retourne la valeur de la ligne suivante par unité de partitionnement. Renvoie null pour la dernière valeur	Information cumulée	oui
count()	Compte le nombre de valeurs trouvées par unité de partitionnement		non
avg()	Calcule la moyenne par unité de partitionnement	Information agrégée pour la moyenne sur toute l'unité de partitionnement Information cumulée pour la moyenne glissante	Non : Pour calculer une moyenne sur toute l'unité de partitionnement Oui : pour calculer une moyenne glissante à l'intérieur de l'unité de partitionnement
mean()	Calcule la moyenne	Information agrégée pour la moyenne sur toute l'unité de partitionnement Information cumulée pour la moyenne glissante	Non : Pour calculer une moyenne sur toute l'unité de partitionnement Oui : pour calculer une moyenne glissante à l'intérieur de l'unité de partitionnement
first() ou first(col, ignorenulls=False)	Renvoie la première valeur trouvée par unité de partitionnement	Information agrégée pour la moyenne sur toute l'unité de partitionnement	non
last() ou last(col, ignorenulls=False)	Renvoie la dernière valeur trouvée par unité de partitionnement	Information agrégée pour la moyenne sur toute l'unité de partitionnement	non
max()	Renvoie la valeur max par unité de partitionnement	Information agrégée pour la moyenne sur toute l'unité de partitionnement	non
min()	Renvoie la valeur min par unité de partitionnement	Information agrégée pour la moyenne sur toute l'unité de partitionnement	non
sum()	Renvoie la somme par unité de partitionnement	Information agrégée pour la somme sur toute l'unité de partitionnement Information cumulée pour la cumulée	Non : Pour calculer une somme sur toute l'unité de partitionnement Oui : pour calculer une somme cumulée ligne par ligne dans l'unité de partitionnement
collect_list()	Renvoie toutes les valeurs trouvées sous forme de listes par unité de partitionnement	Information agrégée pour la moyenne sur toute l'unité de partitionnement	non
collect_set()	Renvoie toutes les valeurs sous forme de set (valeurs distinctes) par unité de partitionnement	Information agrégée pour la moyenne sur toute l'unité de partitionnement	non

4.6.30.1 Cas 1 : Ajouter une information cumulée: la fonction Window().partitionBy() avec l'option orderBy()

On utilise la fonction Window().partitionBy() avec l'option orderBy() lorsqu'on souhaite appliquer les fonctions windows standard sur les colonnes d'un dataframe afin d'ajouter les nouvelles colonnes au dataframe. L'exemple ci-dessous montre l'utilisation des principales fonctions windows utilisables avec l'option orderBy(). Les fonctions windows utilisables avec l'option orderBy() renvoient toujours des informations cumulées. Le cumul de l'information se fait à l'intérieur de l'unité de partitionnement de sorte que la valeur sur la ligne courante est obtenue par incrémentation (arithmétique ou géométrique) de la valeur sur la ligne précédente. Voir exemple ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal',None,'Martin','1995','M',4500),
  ('Pierre','Rose','','1998','M',2000),
  ('Laurent','','Bernard','1995','M',3000),
  ('Virginie',None,'Carpentier','1998','F',2500),
  ('Gabriel','Jonathan','Dubois','1996','M',3200),
  ('Jenny','Carole','Bouvier','1998','F',3500),
  ('Emma',None,'Leroux','1995','F',2800),
  ('Claude','Louis','Morel','1996','M',3100)
  ]

columns = ["prenom","surnom","nom","annee_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

########### Spécification window  avec l'option orderBy() ##############
w = Window.partitionBy("genre").orderBy(F.col("salaire").asc())

""" Autres templates de spécification de windows
# Template 1 : pour partionner suivant plusieurs colonnes
w =
Window.partitionBy("genre","annee_naiss").orderBy(F.col("salaire").asc())
# Template 2 : pour ordonner suivant plusieurs colonnes
w = Window.partitionBy("genre").orderBy(F.col("salaire").asc(),
F.col("annee_naiss").desc()))
"""

# Appliquer les fonctions compatibles avec l'option orderBy() :
```

```
df1=df.withColumn("row_num",F.row_number().over(w))\
        .withColumn("row_rank",F.rank().over(w))\
          .withColumn("lag_an",F.lag("annee_naiss").over(w))\
            .withColumn("lead_an",F.lead("annee_naiss").over(w))
df1.show(truncate=False)
```

```
# Output show() df
+--------+--------+----------+-----------+-----+-------+
|prenom  |surnom  |nom       |annee_naiss|genre|salaire|
+--------+--------+----------+-----------+-----+-------+
|Pascal  |null    |Martin    |1995       |M    |4500   |
|Pierre  |Rose    |          |1998       |M    |2000   |
|Laurent |        |Bernard   |1995       |M    |3000   |
|Virginie|null    |Carpentier|1998       |F    |2500   |
|Gabriel |Jonathan|Dubois    |1996       |M    |3200   |
|Jenny   |Carole  |Bouvier   |1998       |F    |3500   |
|Emma    |null    |Leroux    |1995       |F    |2800   |
|Claude  |Louis   |Morel     |1996       |M    |3100   |
+--------+--------+----------+-----------+-----+-------+

# Output show() df1
+--------+--------+----------+-----------+-----+-------+-------+--------+------+-------+
|prenom  |surnom  |nom       |annee_naiss|genre|salaire|row_num|row_rank|lag_an|lead_an|
+--------+--------+----------+-----------+-----+-------+-------+--------+------+-------+
|Virginie|null    |Carpentier|1998       |F    |2500   |1      |1       |null  |1995   |
|Emma    |null    |Leroux    |1995       |F    |2800   |2      |2       |1998  |1998   |
|Jenny   |Carole  |Bouvier   |1998       |F    |3500   |3      |3       |1995  |null   |
|Pierre  |Rose    |          |1998       |M    |2000   |1      |1       |null  |1995   |
|Laurent |        |Bernard   |1995       |M    |3000   |2      |2       |1998  |1996   |
|Claude  |Louis   |Morel     |1996       |M    |3100   |3      |3       |1995  |1996   |
|Gabriel |Jonathan|Dubois    |1996       |M    |3200   |4      |4       |1996  |1995   |
|Pascal  |null    |Martin    |1995       |M    |4500   |5      |5       |1996  |null   |
+--------+--------+----------+-----------+-----+-------+-------+--------+------+-------+
```

4.6.30.2 Cas 2 : Ajouter une information agrégée : la fonction Window().partitionBy() sans l'option orderBy()

On utilise la fonction Window().partitionBy() sans l'option orderBy() lorsqu'on souhaite appliquer les fonctions d'agrégation standard sur les colonnes d'un dataframe afin d'ajouter les nouvelles colonnes au dataframe. L'exemple ci-dessous montre l'utilisation de quelques fonctions d'agrégation standards avec la fonction Window().partitionBy() sans l'option orderBy(). Rappelons une fois de plus que lorsqu'on applique les fonctions d'agrégation standards sur un dataframe groupé par la fonction groupBy() via la fonction agg(), on obtient un dataframe contenant quelques lignes ; c'est-à-dire une ligne par clé de groupage. A l'inverse lorsqu'on applique ces mêmes fonctions d'agrégation sur un dataframe partitionné par la fonction partitionBy() via la fonction Window(), le dataframe obtenue conserve le même nombre de lignes que le dataframe d'origine et rajoute les informations

agrégation qui sont ajoutées comme colonnes supplémentaires au dataframe. Voir exemple ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal',None,'Martin','1995','M',4500),
  ('Pierre','Rose','','1998','M',2000),
  ('Laurent','','Bernard','1995','M',3000),
  ('Virginie',None,'Carpentier','1998','F',2500),
  ('Gabriel','Jonathan','Dubois','1996','M',3200),
  ('Jenny','Carole','Bouvier','1998','F',3500),
  ('Emma',None,'Leroux','1995','F',2800),
  ('Claude','Louis','Morel','1996','M',3100)
  ]

columns = ["prenom","surnom","nom","annee_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

########### Spécification window  sans l'option orderBy() ##############
w = Window.partitionBy("genre")

""" Autre template de spécification de windows
# Template: pour partionner suivant plusieurs colonnes
w = Window.partitionBy("genre","annee_naiss")
"""

# Appliquer les fonctions valables sans l'option orderBy() :
df1=df.withColumn("count_slr",F.count("salaire").over(w))\
        .withColumn("mean_slr",F.mean("salaire").over(w))\
          .withColumn("min_slr",F.min("salaire").over(w))\
            .withColumn("max_slr",F.max("salaire").over(w))\
              .withColumn("sum_slr",F.sum("salaire").over(w))
df1.show(truncate=False)
```

```
# Output show() df
+--------+--------+----------+-----------+-----+-------+
|prenom  |surnom  |nom       |annee_naiss|genre|salaire|
+--------+--------+----------+-----------+-----+-------+
|Pascal  |null    |Martin    |1995       |M    |4500   |
|Pierre  |Rose    |          |1998       |M    |2000   |
|Laurent |        |Bernard   |1995       |M    |3000   |
|Virginie|null    |Carpentier|1998       |F    |2500   |
```

```
|Gabriel |Jonathan|Dubois    |1996        |M    |3200   |
|Jenny   |Carole  |Bouvier   |1998        |F    |3500   |
|Emma    |null    |Leroux    |1995        |F    |2800   |
|Claude  |Louis   |Morel     |1996        |M    |3100   |
+--------+--------+----------+------------+-----+-------+

# Output show() df1
+--------+--------+----------+-----------+-----+-------+---------+--------+-------+-------+-------+
|prenom  |surnom  |nom       |annee_naiss|genre|salaire|count_slr|mean_slr|min_slr|max_slr|sum_slr|
+--------+--------+----------+-----------+-----+-------+---------+--------+-------+-------+-------+
|Virginie|null    |Carpentier|1998       |F    |2500   |3        |2933.333|2500   |3500   |8800   |
|Emma    |null    |Leroux    |1995       |F    |2800   |3        |2933.333|2500   |3500   |8800   |
|Jenny   |Carole  |Bouvier   |1998       |F    |3500   |3        |2933.333|2500   |3500   |8800   |
|Laurent |        |Bernard   |1995       |M    |3000   |5        |3160.0  |2000   |4500   |15800  |
|Claude  |Louis   |Morel     |1996       |M    |3100   |5        |3160.0  |2000   |4500   |15800  |
|Pierre  |Rose    |          |1998       |M    |2000   |5        |3160.0  |2000   |4500   |15800  |
|Pascal  |null    |Martin    |1995       |M    |4500   |5        |3160.0  |2000   |4500   |15800  |
|Gabriel |Jonathan|Dubois    |1996       |M    |3200   |5        |3160.0  |2000   |4500   |15800  |
+--------+--------+----------+-----------+-----+-------+---------+--------+-------+-------+-------+
```

4.6.30.3 Cas 3 : Ajouter une information cumulée ou agrégée au niveau de tout le dataframe : la fonction Window().partitionBy(F.lit(1)).orderBy(F.lit(1))

Comme on peut le constater à travers les cas1 et cas 2 précédemment présentés, les informations cumulées et les informations agrégées sont généralement calculées au niveau de l'unité de partitionnement définie par partitionBy("col") où "col" représente la colonne de partitionnement. Mais parfois, il arrive qu'on ait besoin d'avoir l'information cumulée/agrégée calculée au niveau de tout le dataframe, et non simplement au niveau chaque clé de partitionnement. C'est le cas par exemple lorsque nous souhaitons calculer la somme cumulée d'une colonne sur tous le dataframe (information cumulée), ou encore lorsqu'on souhaite obtenir la somme totale d'une colonne sur tout le dataframe. Pour gérer ce genre de cas, nous générons une colonne de partitionnement fictive avec une valeur unique ; de même nous générons une colonne de tri –orderBy() fictive également à valeur unique. Ces informations fictives peuvent être générée simplement avec la spécification F.lit(1)). En utilisant ainsi ces colonnes de partitionnement et de tri fictives, on peut indirectement calculer l'information souhaitée. L'exemple ci-dessous montre le calcul des informations agrégées sur le tout le dataframe en utilisant une colonne de partitionnement et tri fictives.

Rappelons toutefois que la procédure décrite ci-dessus est juste une astuce pour calculer les informations souhaitées étant données que la fonction partitionBy() doit être obligatoirement renseignée lorsqu'on utilise la fonctionnalité Window.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
```

```
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('Pascal',None,'Martin','1995','M',4500),
  ('Pierre','Rose','','1998','M',2000),
  ('Laurent','','Bernard','1995','M',3000),
  ('Virginie',None,'Carpentier','1998','F',2500),
  ('Gabriel','Jonathan','Dubois','1996','M',3200),
  ('Jenny','Carole','Bouvier','1998','F',3500),
  ('Emma',None,'Leroux','1995','F',2800),
  ('Claude','Louis','Morel','1996','M',3100)
  ]

columns = ["prenom","surnom","nom","annee_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

########### Spécification window  avec une colonne de partitionnement et de
tri fictives
w=Window.partitionBy(F.lit(1)).orderBy(F.lit(1))

df1=df.withColumn("all_count",F.count("salaire").over(w))\
        .withColumn("all_mean",F.mean("salaire").over(w))\
          .withColumn("all_min",F.min("salaire").over(w))\
            .withColumn("all_max",F.max("salaire").over(w))\
              .withColumn("all_sum",F.sum("salaire").over(w))
df1.show(truncate=False)
```

```
# Output show() df
+--------+--------+----------+-----------+-----+-------+
|prenom  |surnom  |nom       |annee_naiss|genre|salaire|
+--------+--------+----------+-----------+-----+-------+
|Pascal  |null    |Martin    |1995       |M    |4500   |
|Pierre  |Rose    |          |1998       |M    |2000   |
|Laurent |        |Bernard   |1995       |M    |3000   |
|Virginie|null    |Carpentier|1998       |F    |2500   |
|Gabriel |Jonathan|Dubois    |1996       |M    |3200   |
|Jenny   |Carole  |Bouvier   |1998       |F    |3500   |
|Emma    |null    |Leroux    |1995       |F    |2800   |
|Claude  |Louis   |Morel     |1996       |M    |3100   |
+--------+--------+----------+-----------+-----+-------+

# Output show() df1
+--------+--------+----------+-----------+-----+-------+---------+--------+-------+-------+-------+
|prenom  |surnom  |nom       |annee_naiss|genre|salaire|all_count|all_mean|all_min|all_max|all_sum|
+--------+--------+----------+-----------+-----+-------+---------+--------+-------+-------+-------+
|Pascal  |null    |Martin    |1995       |M    |4500   |8        |3075.0  |2000   |4500   |24600  |
|Virginie|null    |Carpentier|1998       |F    |2500   |8        |3075.0  |2000   |4500   |24600  |
|Emma    |null    |Leroux    |1995       |F    |2800   |8        |3075.0  |2000   |4500   |24600  |
|Laurent |        |Bernard   |1995       |M    |3000   |8        |3075.0  |2000   |4500   |24600  |
|Jenny   |Carole  |Bouvier   |1998       |F    |3500   |8        |3075.0  |2000   |4500   |24600  |
|Claude  |Louis   |Morel     |1996       |M    |3100   |8        |3075.0  |2000   |4500   |24600  |
|Pierre  |Rose    |          |1998       |M    |2000   |8        |3075.0  |2000   |4500   |24600  |
|Gabriel |Jonathan|Dubois    |1996       |M    |3200   |8        |3075.0  |2000   |4500   |24600  |
+--------+--------+----------+-----------+-----+-------+---------+--------+-------+-------+-------+
```

4.6.31 Calculer une valeur agrégée par ligne sur plusieurs colonnes du dataframe : somme par ligne, moyenne par ligne, le min et le max par ligne : les fonctions least() et greatest()

Il arrive souvent de rencontrer des situations où nous souhaitons obtenir une information agrégée par ligne sur plusieurs colonnes du dataframe. Par exemples, calculer la somme par ligne (row wise sum), la moyenne par ligne (row wise mean), la min ou le max par ligne (row wise min/max). Spark offre des fonctionnalités pour calculer certaines de ces statistiques mais dans la plupart des cas c'est à l'utilisateur de spécifier lui-même la formule de calcul pour obtenir l'information souhaitée. Par exemple, pour obtenir le min et le max par ligne, on peut utiliser respectivement la fonction least() et la fonction greatest(). En revanche pour calculer la somme par ligne ou la moyenne par ligne, l'utilisateur doit spécifier lui-même l'opération arithmétique sur les colonnes concernées pour obtenir la statistique voulu. L'exemple ci-dessous quelques exemples pour obtenir les informations agrégées par lignes.

Attention tout de même à l'utilisation des opérateurs arithmétiques pour calculer la somme ou la moyenne par ligne. En effet, en faisant par exemple l'addition des colonnes, lorsque la valeur d'une des colonnes additionnées est nulle, la somme calculée sera également. Dans cette situation, il est préférable de gérer le cas des valeurs nulles avant d'appliquer l'opération arithmétique sur les colonnes. Par exemple pour le cas de la somme, on peut d'abord remplacer les valeurs nulles par 0. Par contre, une telle approche n'est pas valide dans le cas du calcul de la moyenne. Car remplacer les valeurs nulles par 0 aboutira à biaiser la valeur moyenne calculée. C'est pourquoi l'utilisation des opérations arithmétiques sur les colonnes doit être faite avec beaucoup de précaution. Remarquons tout de même que l'utilisation native spark telles que least() ou greatest() excluent d'office les valeurs nulles afin de calculer le min et le max sur les colonnes. Par conséquent le résultat renvoyé par n'est pas null lorsqu'au moins il existe une colonne dans le liste qui n'est pas nulle.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [("01",60, 55, 45, 75)
        ,("02", 30,45, 31, 39)
        ,("03",31,None,74,80)
        ,("04",25,45,90,10)
        ,("05", 31,28,None,94)
```

```
        ,("06", 56,62,83,None)
        ]
columns=["id","score1","score2","score3","score4"]
df=spark.createDataFrame(data=data,schema=columns)
df.show()
# Calculer la somme, la moyenne, le min et le max sur les colonnes
df1=df.withColumn("sum",F.col("score1")+F.col("score2")+F.col("score3")+F.c
ol("score4"))\

.withColumn("mean",(F.col("score1")+F.col("score2")+F.col("score3")+F.col("
score4"))/4)\

.withColumn("min",F.least(F.col("score1"),F.col("score2"),F.col("score3"),F
.col("score4")))\

.withColumn("max",F.greatest(F.col("score1"),F.col("score2"),F.col("score3"
),F.col("score4")))
df1.show()
```

```
# Output show() df
+---+------+------+------+------+
| id|score1|score2|score3|score4|
+---+------+------+------+------+
| 01|    60|    55|    45|    75|
| 02|    30|    45|    31|    39|
| 03|    31|  null|    74|    80|
| 04|    25|    45|    90|    10|
| 05|    31|    28|  null|    94|
| 06|    56|    62|    83|  null|
+---+------+------+------+------+

# Output show() df1
+---+------+------+------+------+----+-----+---+---+
| id|score1|score2|score3|score4| sum| mean|min|max|
+---+------+------+------+------+----+-----+---+---+
| 01|    60|    55|    45|    75| 235|58.75| 45| 75|
| 02|    30|    45|    31|    39| 145|36.25| 30| 45|
| 03|    31|  null|    74|    80|null| null| 31| 80|
| 04|    25|    45|    90|    10| 170| 42.5| 10| 90|
| 05|    31|    28|  null|    94|null| null| 28| 94|
| 06|    56|    62|    83|  null|null| null| 56| 83|
+---+------+------+------+------+----+-----+---+---+
```

4.6.32 Renvoyer la première valeur non nulle entre plusieurs colonnes sur une ligne du dataframe : la fonction coalesce()

La fonction coalesce() de transformation (à ne pas confondre avec la fonction coalesce() de repartitionnement du dataframe) permet de renvoyer la première valeur non nulle dans une liste de colonnes. L'exemple ci-dessous montre l'utilisation de la fonction coalesce(). L'exemple ci-dessous montre l'utilisation de la fonction coalesce()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
```

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [("01",60,None, 45, 75)
        ,("02", 30,45,None, 39)
        ,("03",None,None,74,80)
        ,("04",25,45,90,10)
        ,("05", None,None,None,94)
        ,("06", 56,62,83,None)
        ]
columns=["id","essai1","essai2","essai3","essai4"]
df=spark.createDataFrame(data=data,schema=columns)
df.show()
df.printSchema()
# Retenir la première valeur non nulle entre essai1, essai2, essai3 et
essai4
df1=df.withColumn("score_retenu",F.coalesce(*["essai1","essai2","essai3","e
ssai4"]))
df1.show()
df1.printSchema()
```

```
# Output show() df
+---+------+------+------+------+
| id|essai1|essai2|essai3|essai4|
+---+------+------+------+------+
| 01|    60|  null|    45|    75|
| 02|    30|    45|  null|    39|
| 03|  null|  null|    74|    80|
| 04|    25|    45|    90|    10|
| 05|  null|  null|  null|    94|
| 06|    56|    62|    83|  null|
+---+------+------+------+------+
# Output show() df1
+---+------+------+------+------+------------+
| id|essai1|essai2|essai3|essai4|score_retenu|
+---+------+------+------+------+------------+
| 01|    60|  null|    45|    75|          60|
| 02|    30|    45|  null|    39|          30|
| 03|  null|  null|    74|    80|          74|
| 04|    25|    45|    90|    10|          25|
| 05|  null|  null|  null|    94|          94|
| 06|    56|    62|    83|  null|          56|
+---+------+------+------+------+------------+
```

4.6.33 Pivoter un Dataframe : la fonction pivot()

La fonction pivot() est une fonction d'agrégation avancée qui permet d'appliquer une rotation sur le dataframe et d'appliquer une fonction d'agrégation.

Le paramétrage de la fonction pivot() exige trois catégories de colonnes : une ou plusieurs colonnes de groupage, une colonne de pivotage et une colonne d'agrégation.

La (les) colonne(s) de groupage sont les colonnes dont les valeurs combinées permet de définir les groupes dans lesquels l'opération d'agrégation est appliquée. Ces colonnes forment la clé d'agrégation.

La colonne de pivotage est la colonne dont les valeurs sont transposées pour générer plusieurs nouvelles colonnes ; chaque nouvelle colonne correspondant à une valeur unique de la colonne d'origine.

La colonne d'agrégation est la colonne sur laquelle est appliquée une fonction d'agrégation : sum(), mean(), max(), min(), first(), etc…

L'exemple ci-dessous illustre l'utilisation de la fonction pivot(). Dans cet exemple, le dataframe initial fournit le combre médailles de gagnés pour catégorie de médailles pour chaque pays. Toutefois, dans le dataframe initial, il y a un enregistrement par catégorie de médaille et par pays. Ce qui aboutit à plusieurs lignes. L'application de la fonction pivot permet de réduire le nombre de lignes. Il y aura juste autant de ligne de catégories de ménage. En revanche, le pivotage aboutit à augmenter le nombre de colonnes. Car la fonction ajoute autant de colonnes que valeurs pays rencontrées dans la colonne pays. Voir les résultats issus de l'exécution de l'exemple d'application.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [    ("or","USA",39),("argent","USA",41),("bronze","USA",33)
          ,("or","FRANCE",10),("argent","FRANCE",12),("bronze","FRANCE",11)
          ,("or","CHINE",38),("argent","CHINE",32),("bronze","CHINE",18)
          ,("or","BRESIL",7),("argent","BRESIL",6),("bronze","BRESIL",8)
          ,("or","KENYA",4),("argent","KENYA",4),("bronze","KENYA",2)
        ]

columns= ["medaille","pays","nombre"]
df = spark.createDataFrame(data = data, schema = columns)
```

```
df.show(truncate=False)

# Pivoter la colonne pays et calculer la somme par médaille
df1 = df.groupBy("medaille").pivot("pays").sum("nombre")
df1.show(truncate=False)
```

```
# Output show() df
+--------+------+------+
|medaille|pays  |nombre|
+--------+------+------+
|or      |USA   |39    |
|argent  |USA   |41    |
|bronze  |USA   |33    |
|or      |FRANCE|10    |
|argent  |FRANCE|12    |
|bronze  |FRANCE|11    |
|or      |CHINE |38    |
|argent  |CHINE |32    |
|bronze  |CHINE |18    |
|or      |BRESIL|7     |
|argent  |BRESIL|6     |
|bronze  |BRESIL|8     |
|or      |KENYA |4     |
|argent  |KENYA |4     |
|bronze  |KENYA |2     |
+--------+------+------+

# Output show() df1
+--------+------+-----+------+-----+---+
|medaille|BRESIL|CHINE|FRANCE|KENYA|USA|
+--------+------+-----+------+-----+---+
|bronze  |8     |18   |11    |2    |33 |
|argent  |6     |32   |12    |4    |41 |
|or      |7     |38   |10    |4    |39 |
+--------+------+-----+------+-----+---+
```

4.6.34 Générer plusieurs lignes à partir d'une colonne contenant une liste de valeurs: la fonction explode_outer()

La fonction explode_outer() permet de générer plusieurs lignes à partir d'une seule ligne d'un dataframe lorsque la colonne spécifiée est de type Array ou map. Les exemples ci-dessous montrent l'utilisation de la fonction explode_outer().

4.6.34.1 Cas 1 : Cas où la colonne d'origine est déjà de type array() ou map()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()


# Création du Dataframe
data=[("01", ["crayon", "stylo", "cahier"]),
      ("02", ["cahier", "cartable"]),
      ("03", ["cahier", "gomme"]),
      ("04", ["cahier", "crayon", "règle"]),
      ("05", [ ]),
      ("06", None),
      ("07", ["cahier"]),
      ("08", ["feutre", "crayon", "règle"])]

schema = StructType([
    StructField("id",StringType(),True),
    StructField("outils",ArrayType(StringType()),True)])

df = spark.createDataFrame(data=data, schema=schema)
df.show()
print("Nb rows df: "+str(df.count()))

## Appliquer le explode_outer()

df1=df.select("id", F.explode_outer("outils").alias("outils_expl")) #
explode sur la colonne outils.
df1.show()
print("Nb rows df1: "+str(df1.count()))
```

```
# Output show() df
+---+--------------------+
| id|              outils|
+---+--------------------+
| 01|[crayon, stylo, c...|
| 02|  [cahier, cartable]|
| 03|     [cahier, gomme]|
| 04|[cahier, crayon, ...|
| 05|                  []|
| 06|                null|
| 07|            [cahier]|
| 08|[feutre, crayon, ...|
+---+--------------------+
# Output Count df :8

# Output show() df1
+---+-----------+
| id|outils_expl|
+---+-----------+
| 01|     crayon|
| 01|      stylo|
| 01|     cahier|
```

```
| 02|     cahier|
| 02|   cartable|
| 03|     cahier|
| 03|      gomme|
| 04|     cahier|
| 04|     crayon|
| 04|      règle|
| 05|       null|
| 06|       null|
| 07|     cahier|
| 08|     feutre|
| 08|     crayon|
| 08|      règle|
+---+-----------+

# Output Count df1 :16
```

4.6.34.2 Cas 2 : Cas où la colonne d'origine est de type string

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json


spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

# # Création du Dataframe
data=[("01", "crayon ; stylo ; cahier"),
      ("02", "cahier ; cartable"),
      ("03", "cahier ; gomme"),
      ("04", "cahier ; crayon ; règle"),
      ("06", "cahier"),
      ("07", "feutre ; crayon ; règle")
      ]

schema = StructType([
    StructField("id",StringType(),True),
    StructField("outils",StringType(),True)])

df = spark.createDataFrame(data=data, schema=schema)
df.show(truncate=False)
print("Nb rows df: "+str(df.count()))

## Appliquer le explode_outer( split())
df1=df.select("id", F.explode_outer(F.split("outils","
;")).alias("outils_expl")) # explode après split de la colonne outils.
df1.show(truncate=False)
print("Nb rows df1: "+str(df1.count()))
```

```
# Output show() df
+---+-----------------------+
|id |outils                 |
+---+-----------------------+
|01 |crayon ; stylo ; cahier|
|02 |cahier ; cartable      |
|03 |cahier ; gomme         |
|04 |cahier ; crayon ; règle|
|06 |cahier                 |
|07 |feutre ; crayon ; règle|
+---+-----------------------+
# Output Count df :6

# Output show() df1
+---+-----------+
|id |outils_expl|
+---+-----------+
|01 |crayon     |
|01 | stylo     |
|01 | cahier    |
|02 |cahier     |
|02 | cartable  |
|03 |cahier     |
|03 | gomme     |
|04 |cahier     |
|04 | crayon    |
|04 | règle     |
|06 |cahier     |
|07 |feutre     |
|07 | crayon    |
|07 | règle     |
+---+-----------+

# Output Count df1 :14
```

Dans le cas 1, la colonne outils est déjà de type array() dans le dataframe. De ce fait, on peut directement appliquer la fonction explode_outer(). Mais dans le cas 2, la colonne outils est initialement de type string dont les valeurs sont distinguées par un séparateur (ici ;). Pour obtenir la valeur en array, on applique la fonction split() sur la valeur. Ainsi, on pourra appliquer la fonction explode_outer() pour générer une ligne par valeur unique.

Notons, par ailleurs, que la fonction explode_outer() est similaire à la fonction explode(). La seule différence tient au fait que la fonction explode_outer() renvoie une valeur nulle lorsque l'array ou le map en entrée est nul ou vide. Alors que dans dans la même situation la fonction explode() renvoie une erreur.

4.6.35 Faire la jointure entre deux dataframes : la fonction join()

La fonction join() permet de faire la jointure entre deux dataframe en spécifiant ou non une clé. La syntaxe de jointure entre deux dataframes est la suivante df1.join(df2, *cles_de_jointure*, *type_jointure*).

Spark offre plusieurs types de jointure. Le tableau ci-dessous présente les jointures les plus couramment utilisées.

Type de jointure	**description**	**Option spark**
INNER JOIN	Renvoie toutes les lignes qui sont communes aux deux dfs.	"inner"
LEFT JOIN	Renvoie toutes les lignes qui sont communes aux deux dfs + les lignes qui viennent uniquement du df de gauche	"left", "leftouter" ou "left_outer"
RIGHT JOIN	Renvoie toutes les lignes qui sont communes aux deux dfs + les lignes qui viennent uniquement du df de droites	"right", "rightouter" ou "right_outer"
FULL OUTER JOIN	Renvoie toutes les lignes qui viennent des deux dfs	"outer", "full", "fullouter" ou "full_outer"
LEFT ANTI	Renvoie toutes les lignes qui viennent uniquement du df de droite.	"anti", "leftanti", "left_anti" NB : Pour obtenir l'option right anti, inverser simplement l'ordre des dfs dans la jointure et appliquer left anti.
LEFT SEMI	Renvoie uniquement les lignes et les colonnes du df de gauche lorsque les lignes sont communes entre le df de gauche et le df de droite.	"semi", "leftsemi", "left_semi" NB : Pour obtenir l'option right anti, inverser simplement l'ordre des dfs dans la jointure et appliquer left anti.
CROSS JOIN	Jointure cartésienne : chaque ligne du dataframe de gauche est jointe à toutes les lignes du dataframe de droite.	crossJoin()

Les exemples ci-dessous montrent l'utilisation de chaque type de jointure.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data1 = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns1 = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df1 = spark.createDataFrame(data=data1, schema = columns1)
df1.show(truncate=False)
data2 = [('01','London' ,10),
        ('02','Paris' ,5),
        ('03','New York' ,4),
      ('06','Tokyo', 6),
        ('07','Pekin', 3)]
columns2 = ["id","ville","anciennete"]
df2 = spark.createDataFrame(data=data2, schema = columns2)
df2.show(truncate=False)

# INNER JOIN

joined_df1=df1.join(df2, on=["id"], how="inner") # spécification 1
""" Autres spécifications de inner join
joined_df1=df1.join(df2, ["id"], "inner") # spécification 2
joined_df1=df1.join(df2, df1["id"]== df1["id"], "inner") # spécification 3
joined_df1=df1.join(df2, df1.id== df1.id, "inner") # spécification 4
"""
joined_df1.sort("id").show(truncate=False)

# LEFT JOIN
joined_df2=df1.join(df2, ["id"], "left")
joined_df2.sort("id").show(truncate=False)
# RIGHT JOIN

joined_df3=df1.join(df2, ["id"], "right")
joined_df3.sort("id").show(truncate=False)

# FULL OUTER JOIN

joined_df4=df1.join(df2, ["id"], "outer")
joined_df4.sort("id").show(truncate=False)

# LEFT ANTI

joined_df5=df1.join(df2, ["id"], "left_anti")
joined_df5.sort("id").show(truncate=False)

# LEFT SEMI
```

```
joined_df6=df1.join(df2, ["id"], "leftsemi")
joined_df6.sort("id").show(truncate=False)

# CROSS JOIN

joined_df7=df1.crossJoin(df2)
joined_df7.drop(df2.id).show(truncate=False)
```

```
# Output show() df1
+---+--------+------+----------+----------+-----+-------+
|id |prenom  |surnom|nom       |date_naiss|genre|salaire|
+---+--------+------+----------+----------+-----+-------+
|01 |Pascal  |      |Martin    |1995-02-25|M    |4500   |
|02 |Pierre  |Rose  |          |1998-07-12|M    |3000   |
|03 |Laurent |      |Bernard   |2000-09-14|M    |3000   |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|05 |Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+---+--------+------+----------+----------+-----+-------+

# Output show() df2
+---+--------+----------+
|id |ville   |anciennete|
+---+--------+----------+
|01 |London  |10        |
|02 |Paris   |5         |
|03 |New York|4         |
|06 |Tokyo   |6         |
|07 |Pekin   |3         |
+---+--------+----------+

# Output show() INNER JOIN  joined_df1

+---+-------+------+-------+----------+-----+-------+--------+----------+
|id |prenom |surnom|nom    |date_naiss|genre|salaire|ville   |anciennete|
+---+-------+------+-------+----------+-----+-------+--------+----------+
|01 |Pascal |      |Martin |1995-02-25|M    |4500   |London  |10        |
|02 |Pierre |Rose  |       |1998-07-12|M    |3000   |Paris   |5         |
|03 |Laurent|      |Bernard|2000-09-14|M    |3000   |New York|4         |
+---+-------+------+-------+----------+-----+-------+--------+----------+

# Output show() LEFT JOIN joined_df2
+---+--------+------+----------+----------+-----+-------+--------+----------+
|id |prenom  |surnom|nom       |date_naiss|genre|salaire|ville   |anciennete|
+---+--------+------+----------+----------+-----+-------+--------+----------+
|01 |Pascal  |      |Martin    |1995-02-25|M    |4500   |London  |10        |
|02 |Pierre  |Rose  |          |1998-07-12|M    |3000   |Paris   |5         |
|03 |Laurent |      |Bernard   |2000-09-14|M    |3000   |New York|4         |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |null    |null      |
|05 |Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |null    |null      |
+---+--------+------+----------+----------+-----+-------+--------+----------+

# Output show() RIGHT JOIN joined_df3
+---+-------+------+-------+----------+-----+-------+--------+----------+
|id |prenom |surnom|nom    |date_naiss|genre|salaire|ville   |anciennete|
+---+-------+------+-------+----------+-----+-------+--------+----------+
|01 |Pascal |      |Martin |1995-02-25|M    |4500   |London  |10        |
|02 |Pierre |Rose  |       |1998-07-12|M    |3000   |Paris   |5         |
|03 |Laurent|      |Bernard|2000-09-14|M    |3000   |New York|4         |
|06 |null   |null  |null   |null      |null |null   |Tokyo   |6         |
|07 |null   |null  |null   |null      |null |null   |Pekin   |3         |
```

```
+---+-------+------+-------+----------+-----+-------+--------+----------+

# Output show() FULL OUTER JOIN joined_df4
+---+--------+------+---------+----------+-----+-------+--------+----------+
|id |prenom  |surnom|nom      |date_naiss|genre|salaire|ville   |anciennete|
+---+--------+------+---------+----------+-----+-------+--------+----------+
|01 |Pascal  |      |Martin   |1995-02-25|M    |4500   |London  |10        |
|02 |Pierre  |Rose  |         |1998-07-12|M    |3000   |Paris   |5         |
|03 |Laurent |      |Bernard  |2000-09-14|M    |3000   |New York|4         |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |null    |null      |
|05 |Jenny   |Carole|Bouvier  |1982-02-15|F    |3500   |null    |null      |
|06 |null    |null  |null     |null      |null |null   |Tokyo   |6         |
|07 |null    |null  |null     |null      |null |null   |Pekin   |3         |
+---+--------+------+---------+----------+-----+-------+--------+----------+

# Output show() LEFT ANTI joined_df5

+---+--------+------+---------+----------+-----+-------+
|id |prenom  |surnom|nom      |date_naiss|genre|salaire|
+---+--------+------+---------+----------+-----+-------+
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|05 |Jenny   |Carole|Bouvier  |1982-02-15|F    |3500   |
+---+--------+------+---------+----------+-----+-------+

# Output show() LEFT SEMI joined_df6
+---+-------+------+-------+----------+-----+-------+
|id |prenom |surnom|nom    |date_naiss|genre|salaire|
+---+-------+------+-------+----------+-----+-------+
|01 |Pascal |      |Martin |1995-02-25|M    |4500   |
|02 |Pierre |Rose  |       |1998-07-12|M    |3000   |
|03 |Laurent|      |Bernard|2000-09-14|M    |3000   |
+---+-------+------+-------+----------+-----+-------+

# Output show() CROSS JOIN joined_df7
+---+--------+------+---------+----------+-----+-------+--------+----------+
|id |prenom  |surnom|nom      |date_naiss|genre|salaire|ville   |anciennete|
+---+--------+------+---------+----------+-----+-------+--------+----------+
|01 |Pascal  |      |Martin   |1995-02-25|M    |4500   |London  |10        |
|01 |Pascal  |      |Martin   |1995-02-25|M    |4500   |Paris   |5         |
|01 |Pascal  |      |Martin   |1995-02-25|M    |4500   |New York|4         |
|01 |Pascal  |      |Martin   |1995-02-25|M    |4500   |Tokyo   |6         |
|01 |Pascal  |      |Martin   |1995-02-25|M    |4500   |Pekin   |3         |
|02 |Pierre  |Rose  |         |1998-07-12|M    |3000   |London  |10        |
|02 |Pierre  |Rose  |         |1998-07-12|M    |3000   |Paris   |5         |
|02 |Pierre  |Rose  |         |1998-07-12|M    |3000   |New York|4         |
|02 |Pierre  |Rose  |         |1998-07-12|M    |3000   |Tokyo   |6         |
|02 |Pierre  |Rose  |         |1998-07-12|M    |3000   |Pekin   |3         |
|03 |Laurent |      |Bernard  |2000-09-14|M    |3000   |London  |10        |
|03 |Laurent |      |Bernard  |2000-09-14|M    |3000   |Paris   |5         |
|03 |Laurent |      |Bernard  |2000-09-14|M    |3000   |New York|4         |
|03 |Laurent |      |Bernard  |2000-09-14|M    |3000   |Tokyo   |6         |
|03 |Laurent |      |Bernard  |2000-09-14|M    |3000   |Pekin   |3         |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |London  |10        |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |Paris   |5         |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |New York|4         |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |Tokyo   |6         |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |Pekin   |3         |
+---+--------+------+---------+----------+-----+-------+--------+----------+
only showing top 20 rows
```

Remarque : Dans le code présenté ci-dessus, lorsqu'on utilise par exemple la spécification 1 et 2, une seule colonne «id » est gardée dans le df de sortie joined_df1. Mais lorsqu'on utilise la spécification 3 ou 4, deux colonnes « id » sont gardées dans le

df de sortie. La première colonne « id » provient de df1 et la colonne « id » provient de df2. C'est pourquoi il est souvent recommandé de renommer la colonne « id » du df2 et de la supprimer après la jointure. Ces opérations se résument comme suit.

```
df2=df2.withColumnRenamed("id","id2")
joined_df1=df1.join(df2, df1["id"]== df1["id2"], "inner").drop("id2")
```

4.6.36 Empiler deux dataframes : la fonction union ()

La fonction union() permet d'empiler deux dataframes en les juxtaposant l'un sur l'autre (append). Contrairement à la fonction join() qui rajoute des nouvelles colonnes et des nouvelles lignes au dataframe de sortie à partir des deux dataframes en entrée, la fonction union() vise principalement à empiler les lignes des deux dataframes en entrée ayant les mêmes colonnes. L'exemple ci-dessous montre l'utilisation de la fonction union().

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data1 = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns1 = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df1 = spark.createDataFrame(data=data1, schema = columns1)
df1.show(truncate=False)

data2 = [('06','Vincent','','Lebron','1998-05-03','M',2000),
  ('07','Victor','Claude','','2005-01-20','M',8000),
  ('08','Martha','','Lopez','1984-02-09','M',6000),
  ('09','Emily','','Bouvier','1959-12-12','F',3000),
  ('10','Stacy','Goldberg','Samantha','2000-02-10','F', 5000)]

columns2 = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df2 = spark.createDataFrame(data=data2, schema = columns2)
df2.show(truncate=False)

# Union des deux dataframes
df3=df1.union(df2)
df3.show(truncate=False)
```

```
# Output show() df1
+---+--------+------+----------+----------+-----+-------+
|id |prenom  |surnom|nom       |date_naiss|genre|salaire|
+---+--------+------+----------+----------+-----+-------+
|01 |Pascal  |      |Martin    |1995-02-25|M    |4500   |
|02 |Pierre  |Rose  |          |1998-07-12|M    |3000   |
|03 |Laurent |      |Bernard   |2000-09-14|M    |3000   |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|05 |Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+---+--------+------+----------+----------+-----+-------+

# Output show() df2
+---+-------+--------+--------+----------+-----+-------+
|id |prenom |surnom  |nom     |date_naiss|genre|salaire|
+---+-------+--------+--------+----------+-----+-------+
|06 |Vincent|        |Lebron  |1998-05-03|M    |2000   |
|07 |Victor |Claude  |        |2005-01-20|M    |8000   |
|08 |Martha |        |Lopez   |1984-02-09|M    |6000   |
|09 |Emily  |        |Bouvier |1959-12-12|F    |3000   |
|10 |Stacy  |Goldberg|Samantha|2000-02-10|F    |5000   |
+---+-------+--------+--------+----------+-----+-------+

# Output show() UNION  df3

+---+--------+--------+----------+----------+-----+-------+
|id |prenom  |surnom  |nom       |date_naiss|genre|salaire|
+---+--------+--------+----------+----------+-----+-------+
|01 |Pascal  |        |Martin    |1995-02-25|M    |4500   |
|02 |Pierre  |Rose    |          |1998-07-12|M    |3000   |
|03 |Laurent |        |Bernard   |2000-09-14|M    |3000   |
|04 |Virginie|Anne    |Carpentier|1989-10-06|F    |2500   |
|05 |Jenny   |Carole  |Bouvier   |1982-02-15|F    |3500   |
|06 |Vincent |        |Lebron    |1998-05-03|M    |2000   |
|07 |Victor  |Claude  |          |2005-01-20|M    |8000   |
|08 |Martha  |        |Lopez     |1984-02-09|M    |6000   |
|09 |Emily   |        |Bouvier   |1959-12-12|F    |3000   |
|10 |Stacy   |Goldberg|Samantha  |2000-02-10|F    |5000   |
+---+--------+--------+----------+----------+-----+-------+
```

4.6.37 Soustraire deux dataframes : la fonction subtract()

La fonction subtract permet de soustraire deux dataframes et renvoie les lignes qui se trouvent dans le dataframe de gauche mais ne se trouvent pas dans le dataframe de droite. La fonction subtract est quelque peu équivalente à l'utilisation de l'option « left_anti » de la fonction join() à la différence que la fonction subtract utilisent les valeurs de toutes les colonnes pour comparer ligne par ligne les deux dataframes. Elle retient au final les lignes du df de gauche qui n'ont pas de correspondant dans le df de droite. L'exemple ci-dessous montre l'utilisation de la fonction subtract().

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
```

```python
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data1 = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns1 = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df1 = spark.createDataFrame(data=data1, schema = columns1)
df1.show(truncate=False)

data2 = [
      ('01','Pascal','','Martin','1995-02-25','M',4500),
      ('02','Pierre','Rose','','1998-07-12','M',3000),
      ('03','Laurent','','Bernard','2000-09-14','M',3000),
      ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
      ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500),
        ('06','Vincent','','Lebron','1998-05-03','M',2000),
      ('07','Victor','Claude','','2005-01-20','M',8000),
      ('08','Martha','','Lopez','1984-02-09','M',6000),
      ('09','Emily','','Bouvier','1959-12-12','F',3000),
      ('10','Stacy','Goldberg','Samantha','2000-02-10','F', 5000)]

columns2 = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df2 = spark.createDataFrame(data=data2, schema = columns2)
df2.show(truncate=False)

# substract des deux dataframes

df3=df2.subtract(df1)
df3.show(truncate=False)
```

```
# Output show() df1
+---+--------+------+----------+----------+-----+-------+
|id |prenom  |surnom|nom       |date_naiss|genre|salaire|
+---+--------+------+----------+----------+-----+-------+
|01 |Pascal  |      |Martin    |1995-02-25|M    |4500   |
|02 |Pierre  |Rose  |          |1998-07-12|M    |3000   |
|03 |Laurent |      |Bernard   |2000-09-14|M    |3000   |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|05 |Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+---+--------+------+----------+----------+-----+-------+

# Output show() df2
+---+--------+--------+----------+----------+-----+-------+
|id |prenom  |surnom  |nom       |date_naiss|genre|salaire|
+---+--------+--------+----------+----------+-----+-------+
|01 |Pascal  |        |Martin    |1995-02-25|M    |4500   |
|02 |Pierre  |Rose    |          |1998-07-12|M    |3000   |
|03 |Laurent |        |Bernard   |2000-09-14|M    |3000   |
```

```
|04 |Virginie|Anne    |Carpentier|1989-10-06|F    |2500   |
|05 |Jenny   |Carole  |Bouvier   |1982-02-15|F    |3500   |
|06 |Vincent |        |Lebron    |1998-05-03|M    |2000   |
|07 |Victor  |Claude  |          |2005-01-20|M    |8000   |
|08 |Martha  |        |Lopez     |1984-02-09|M    |6000   |
|09 |Emily   |        |Bouvier   |1959-12-12|F    |3000   |
|10 |Stacy   |Goldberg|Samantha  |2000-02-10|F    |5000   |
+---+--------+--------+----------+----------+-----+-------+

# Output show() subtract df3

+---+-------+--------+--------+----------+-----+-------+
|id |prenom |surnom  |nom     |date_naiss|genre|salaire|
+---+-------+--------+--------+----------+-----+-------+
|06 |Vincent|        |Lebron  |1998-05-03|M    |2000   |
|07 |Victor |Claude  |        |2005-01-20|M    |8000   |
|08 |Martha |        |Lopez   |1984-02-09|M    |6000   |
|09 |Emily  |        |Bouvier |1959-12-12|F    |3000   |
|10 |Stacy  |Goldberg|Samantha|2000-02-10|F    |5000   |
+---+-------+--------+--------+----------+-----+-------+
```

4.6.38 Appliquer une fonction-utilisateur sur les colonnes d'un dataframe : la fonction udf()

La fonction udf() permet d'appliquer sur les colonnes d'un dataframe une fonction de traitement personnalisé que l'utilisateur a lui-même conçu et implémenté. Les UDF (User Defined Functions) permettent d'étendre les capacités de Spark, car elles permettent de définir les fonctions de traitement non disponibles dans le package de base de Spark. L'exemple ci-dessus illustre la définition et l'application d'une UDF sur un Dataframe Spark. L'exercice consiste à normaliser les valeur d'une colonne de String, c'est-à-dire éliminer les accents et les caractères spéciaux.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json
import unicodedata

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
data = [('01','Martin','Maracaibó'),
  ('02','Rose','São Paulo'),
  ('03','Bernard','Mad rid'),
  ('04','Carpentier','Paris'),
  ('05','Bouvier','Londo n')]

columns = ["id","nom","ville"]
```

```
df1 = spark.createDataFrame(data=data, schema = columns)
df1.show()

# Définir une UDF pour normaliser une colonne en string: supprimer les
caractères spéciaux du string.
def string_normalizor(str_value):
      str_value = str_value.replace('\n', ' ').replace('\r', '').strip()
      norm_val = unicodedata.normalize('NFKD', str_value).encode('ascii',
'ignore').decode("utf8")
      return(norm_val)

# Convertir la fonction en UDF"""
string_normalize = F.udf(lambda x: string_normalizor(x),StringType())

# Normalier la valeur de la colonne city
df1=df1.withColumn("ville",string_normalize(F.col("ville")))
df1.show(truncate=False)
```

```
# Output show() df
+---+----------+---------+
| id|       nom|    ville|
+---+----------+---------+
| 01|    Martin|Maracaibó|
| 02|      Rose|São Paulo|
| 03|   Bernard|  Madrid|
| 04|Carpentier|    Paris|
| 05|   Bouvier|  London|
+---+----------+---------+

# Output show() df2
+---+----------+---------+
|id |nom       |ville    |
+---+----------+---------+
|01 |Martin    |Maracaibo|
|02 |Rose      |Sao Paulo |
|03 |Bernard   |Madrid   |
|04 |Carpentier|Paris    |
|05 |Bouvier   |London   |
+---+----------+---------+
```

4.6.39 Traiter les colonnes de type date, timestamp et epoch dans un Dataframe: les fonctions current_date(), current_timestamp(), to_date(), to_timestamps(), date_format(), unix_timestamp(), from_unixtime() et les fonctions dérivées

PySpark offre de nombreuses fonctions pour traiter les colonnes de type date et timestamps dans un Dataframe. Les plus fréquemment rencontrées sont présentées dans le tableau ci-dessous.

Fonction	Description
current_date()	Calcule la date du jour

current_timestamp()	Renvoie le timestamp courant
to_date()	Convertit une valeur d'entrée en date
to_timestamps()	Convertit une valeur d'entrée en timestamp
unix_timestamp()	Renvoie l'epoch, le nombre de secondes écoulées depuis 1970-01-01 00:00:00 jusqu'à la date indiquée en paramètre
from_unixtime()	renvoie le timestamp à partir de l'epoch spécifié en paramètre
datediff()	Calcule la différence entre deux dates en nombre de jours
months_between()	Renvoie le nombre de mois entre deux dates
add_months()	Ajoute un certain nombre de mois à une date donnée
date_add()	Ajoute un certain nombre de jours à une date donnée
date_sub()	Soustrait un certain nombre de jours d'une date donnée
last_day()	Renvoie le dernier jour du mois à partir de la date spécifiée

Les exemples ci-dessous illustrent l'utilisation des fonctions usuelles de dates sur un dataframe.

4.6.39.1 Générer la date courante ou le timestamp courant : current_date() et current_timestamp()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json


spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
# Créer un dataframe vide

data = [(3.94,),(4.55,),(0.8,),(2.87,),(4.2,)]

columns = ["record",]
df = spark.createDataFrame(data,columns)
df.show(truncate=False)
df.printSchema()
# Créer deux colonnes date_record, timestamp_record
df1=df.withColumn("date_record", F.current_date()) # date courante
df1=df1.withColumn("timestamp_record", F.current_timestamp()) # timestamp
courant
df1.show(truncate=False)
df1.printSchema()
```

```
# Output show() df
+------+
|record|
```

```
+------+
|3.94  |
|4.55  |
|0.8   |
|2.87  |
|4.2   |
+------+
# output printSchema df
root
 |-- record: double (nullable = true)

# Output show() df2
+------+-----------+-----------------------+
|record|date_record|timestamp_record       |
+------+-----------+-----------------------+
|3.94  |2022-05-29 |2022-05-29 09:52:31.391|
|4.55  |2022-05-29 |2022-05-29 09:52:31.391|
|0.8   |2022-05-29 |2022-05-29 09:52:31.391|
|2.87  |2022-05-29 |2022-05-29 09:52:31.391|
|4.2   |2022-05-29 |2022-05-29 09:52:31.391|
+------+-----------+-----------------------+
root
| -- record: double(nullable=true)
| -- date_record: date(nullable=false)
| -- timestamp_record: timestamp(nullable=false)
```

4.6.39.2 Convertir un string en date ou en timestamp : les fonctions to_date() et to_timestamp()

Les fonctions to_date() et to_timestamp() permettent de convertir une colonne de type string en une colonne de type date ou timestamp. Les exemples ci-dessous montrent l'utilisation des deux fonctions.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
# Créer un dataframe
data = [(3.94,"2022-05-20","2022-05-20 11:01:19.06"),
        (4.55,"2022-05-20","2022-05-20 11:01:19.06"),
        (0.8,"2022-05-20","2022-05-20 11:01:19.06"),
        (2.87,"2022-05-20","2022-05-20 11:01:19.06"),
        (4.2,"2022-05-20","2022-05-20 11:01:19.06")
       ]

columns = ["record","date_record","timestamp_record"]
df = spark.createDataFrame(data,columns)
df.show(truncate=False)
```

```
df.printSchema()

# Convertir les colonnes date_record, timestamp_record en date et timestam
df1=df.withColumn("date_record", F.to_date(F.col("date_record"),"yyyy-MM-
dd"))
df1=df1.withColumn("timestamp_record",
F.to_timestamp(F.col("timestamp_record"),"yyyy-MM-dd HH:mm:ss.SS"))
df1.show(truncate=False)
df1.printSchema()
```

```
# Output show() df
+------+-----------+----------------------+
|record|date_record|timestamp_record      |
+------+-----------+----------------------+
|3.94  |2022-05-20 |2022-05-20 11:01:19.06|
|4.55  |2022-05-20 |2022-05-20 11:01:19.06|
|0.8   |2022-05-20 |2022-05-20 11:01:19.06|
|2.87  |2022-05-20 |2022-05-20 11:01:19.06|
|4.2   |2022-05-20 |2022-05-20 11:01:19.06|
+------+-----------+----------------------+
# output printSchema df
root
 |-- record: double (nullable = true)
 |-- date_record: string (nullable = true)
 |-- timestamp_record: string (nullable = true)

# Output show() df2
+------+-----------+-------------------+
|record|date_record|timestamp_record   |
+------+-----------+-------------------+
|3.94  |2022-05-20 |2022-05-20 11:01:19|
|4.55  |2022-05-20 |2022-05-20 11:01:19|
|0.8   |2022-05-20 |2022-05-20 11:01:19|
|2.87  |2022-05-20 |2022-05-20 11:01:19|
|4.2   |2022-05-20 |2022-05-20 11:01:19|
+------+-----------+-------------------+
root
 |-- record: double (nullable = true)
 |-- date_record: date (nullable = true)
 |-- timestamp_record: timestamp (nullable = true)
```

4.6.39.3 Convertir une colonne date ou timestamp en string : la fonction date_format()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
```

```
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [(3.94,),(4.55,),(0.8,),(2.87,),(4.2,) ]
columns = ["record"]
df = spark.createDataFrame(data,columns)

# Generer d'abord deux colonnes date_record, timestamp_record en format
date et timestamp: current_date() et current_timestamp
df1=df.withColumn("date_record", F.current_date())
df1=df1.withColumn("timestamp_record", F.current_timestamp())
df1.show(truncate=False)
df1.printSchema()
## Convertir les deux colonnes en string
df2=df1.withColumn("date_string", F.date_format("date_record","yyyy-MM-
dd"))
df2=df2.withColumn("timestamp_string",
F.date_format("timestamp_record","yyyy-MM-dd HH:mm:ss.SSS "))
df2.show(truncate=False)
df2.printSchema()
```

```
# Output show() df1
+------+-----------+-----------------------+
|record|date_record|timestamp_record       |
+------+-----------+-----------------------+
|3.94  |2022-05-29 |2022-05-29 03:37:35.646|
|4.55  |2022-05-29 |2022-05-29 03:37:35.646|
|0.8   |2022-05-29 |2022-05-29 03:37:35.646|
|2.87  |2022-05-29 |2022-05-29 03:37:35.646|
|4.2   |2022-05-29 |2022-05-29 03:37:35.646|
+------+-----------+-----------------------+
# output printSchema df1
root
 |-- record: double (nullable = true)
 |-- date_record: date (nullable = false)
 |-- timestamp_record: timestamp (nullable = false)

# Output show() df2
+------+-----------+-----------------------+-----------+-----------------------+
|record|date_record|timestamp_record       |date_string|timestamp_string       |
+------+-----------+-----------------------+-----------+-----------------------+
|3.94  |2022-05-29 |2022-05-29 03:37:37.858|2022-05-29 |2022-05-29 03:37:37.858 |
|4.55  |2022-05-29 |2022-05-29 03:37:37.858|2022-05-29 |2022-05-29 03:37:37.858 |
|0.8   |2022-05-29 |2022-05-29 03:37:37.858|2022-05-29 |2022-05-29 03:37:37.858 |
|2.87  |2022-05-29 |2022-05-29 03:37:37.858|2022-05-29 |2022-05-29 03:37:37.858 |
|4.2   |2022-05-29 |2022-05-29 03:37:37.858|2022-05-29 |2022-05-29 03:37:37.858 |
+------+-----------+-----------------------+-----------+-----------------------+
root
 |-- record: double (nullable = true)
 |-- date_record: date (nullable = false)
 |-- timestamp_record: timestamp (nullable = false)
 |-- date_string: string (nullable = false)
 |-- timestamp_string: string (nullable = false)
```

4.6.39.4 Convertir une date ou un timestamp() en epoch (en nombre de secondes) : la fonction unix_timestamp()

La fonction unix_timestamp() permet de calculer le nombre de secondes écoulées depuis le 1er Janvier 1970 à minuit jusqu'à la date spécifiée en paramètre. Les exemples ci-dessous montrent quelques utilisations de la fonction unix_timestamp()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [(3.94,"2022-05-20","2022-05-20 11:01:19.06"),
      (4.55,"2022-05-20","2022-05-20 11:01:19.06"),
      (0.8,"2022-05-20","2022-05-20 11:01:19.06"),
      (2.87,"2022-05-20","2022-05-20 11:01:19.06"),
      (4.2,"2022-05-20","2022-05-20 11:01:19.06")
       ]

columns = ["record","date_record","timestamp_record"]
df = spark.createDataFrame(data,columns)
# Mettre les deux input colonnes date_record, timestamp_record en format
date et timestamp
df1=df.withColumn("date_record", F.to_date("date_record","yyyy-MM-dd"))
df1=df1.withColumn("timestamp_record",
F.to_timestamp("timestamp_record","yyyy-MM-dd HH:mm:ss.SSS"))
df1.show(truncate=False)
df1.printSchema()
# Calculer les epoch à partir des colonnes timestamp
df2=df1.withColumn("epoch_from_dt", F.unix_timestamp("date_record","yyyy-
MM-dd"))
df2=df2.withColumn("epoch_from_ts",
F.unix_timestamp("timestamp_record","yyyy-MM-dd HH:mm:ss.SSS"))
df2=df2.withColumn("epoch_from_cts", F.unix_timestamp()) # calcul par
defaut à partir de current_timestamp() avec le format par défaut "yyyy-MM-
dd HH:mm:ss"
df2.show(truncate=False)
df2.printSchema()
```

```
# Output show() df1
+------+-----------+-------------------+
|record|date_record|timestamp_record   |
+------+-----------+-------------------+
```

```
|3.94  |2022-05-20 |2022-05-20 11:01:19|
|4.55  |2022-05-20 |2022-05-20 11:01:19|
|0.8   |2022-05-20 |2022-05-20 11:01:19|
|2.87  |2022-05-20 |2022-05-20 11:01:19|
|4.2   |2022-05-20 |2022-05-20 11:01:19|
+------+-----------+-------------------+
# output printSchema df1
root
 |-- record: double (nullable = true)
 |-- date_record: date (nullable = true)
 |-- timestamp_record: timestamp (nullable = true)

# Output show() df2
+------+-----------+-------------------+-------------+-------------+--------------+
|record|date_record|timestamp_record   |epoch_from_dt|epoch_from_ts|epoch_from_cts|
+------+-----------+-------------------+-------------+-------------+--------------+
|3.94  |2022-05-20 |2022-05-20 11:01:19|1652997600   |1653037279   |1653789489    |
|4.55  |2022-05-20 |2022-05-20 11:01:19|1652997600   |1653037279   |1653789489    |
|0.8   |2022-05-20 |2022-05-20 11:01:19|1652997600   |1653037279   |1653789489    |
|2.87  |2022-05-20 |2022-05-20 11:01:19|1652997600   |1653037279   |1653789489    |
|4.2   |2022-05-20 |2022-05-20 11:01:19|1652997600   |1653037279   |1653789489    |
+------+-----------+-------------------+-------------+-------------+--------------+

# output printSchema df1
root
 |-- record: double (nullable = true)
 |-- date_record: date (nullable = true)
 |-- timestamp_record: timestamp (nullable = true)
 |-- epoch_from_dt: long (nullable = true)
 |-- epoch_from_ts: long (nullable = true)
 |-- epoch_from_cts: long (nullable = true)
```

4.6.39.5 Convertir une epoch (nombre secondes) en une date ou un timestamp() : la fonction from_unixtime()

La fonction from_unixtime() permet de convertir une epoch (nombre de secondes) en date ou timestamp. Les exemples ci-dessous illustrent l'utilisation de la fonction.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [(3.94, 1653135707 ),
        (4.55, 1653135752),
        (0.8, 1653135762),
        (2.87, 1653135773),
        (4.2, 1653135781)
```

```
            ]

columns = ["record","record_epoch"]
df = spark.createDataFrame(data,columns)
df.show(truncate=False)
df.printSchema()
# Créer deux colonnes date_record, timestamp_record à partir de l'epoch
df1=df.withColumn("date_record", F.from_unixtime("record_epoch","yyyy-MM-
dd"))
df1=df1.withColumn("timestamp_record",
F.from_unixtime("record_epoch","yyyy-MM-dd HH:mm:ss.SSS"))
df1.show(truncate=False)
df1.printSchema()
```

```
# Output show() df
+------+------------+
|record|record_epoch|
+------+------------+
|3.94  |1653135707  |
|4.55  |1653135752  |
|0.8   |1653135762  |
|2.87  |1653135773  |
|4.2   |1653135781  |
+------+------------+
# output printSchema df
root
 |-- record: double (nullable = true)
 |-- record_epoch: long (nullable = true)

# Output show() df1
+------+------------+-----------+-----------------------+
|record|record_epoch|date_record|timestamp_record       |
+------+------------+-----------+-----------------------+
|3.94  |1653135707  |2022-05-21 |2022-05-21 14:21:47.000|
|4.55  |1653135752  |2022-05-21 |2022-05-21 14:22:32.000|
|0.8   |1653135762  |2022-05-21 |2022-05-21 14:22:42.000|
|2.87  |1653135773  |2022-05-21 |2022-05-21 14:22:53.000|
|4.2   |1653135781  |2022-05-21 |2022-05-21 14:23:01.000|
+------+------------+-----------+-----------------------+
# output printSchema df1
root
 |-- record: double (nullable = true)
 |-- record_epoch: long (nullable = true)
 |-- date_record: string (nullable = true)
 |-- timestamp_record: string (nullable = true)
```

4.6.39.6 Les autres fonctions usuelles de traitement de date : datediff(), months_between(), add_months(), date_add(), date_sub() et last_day()

Les exemples ci-dessous montrent l'utilisation des fonctions dérivées de traitement de date :

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [(3.94,"2021-10-05"),(4.55,"2021-10-05"), (0.8,"2021-10-
05"),(2.87,"2021-10-05")]

columns = ["record","dt"]
df = spark.createDataFrame(data,columns)
# Convertir dt en date
df=df.withColumn("dt", F.col("dt").cast("date"))
df.show(truncate=False)
df.printSchema()
# datediff()
df1=df.withColumn("nb_jours", F.datediff(F.current_date(), F.col("dt"))) #
Nombre de jours depuis "2021-10-05
# months_between()
df1=df1.withColumn("nb_month", F.months_between(F.current_date(),
F.col("dt"))) # Nombre de mois depuis "2021-10-05
# add_months()
df1=df1.withColumn("date_in_8months", F.add_months(F.current_date(), 8)) #
Ajouter 8 mois à la date
# date_add()
df1=df1.withColumn("date_in_20days", F.date_add(F.current_date(), 20)) #
ajouter 20 jours à la date
# date_sub()
df1=df1.withColumn("date_on_20days_back", F.date_sub(F.current_date(), 20))
# soustraire 20 jours de la date
# year()
df1=df1.withColumn("current_year", F.year(F.current_date())) # Année
courante
# month()
df1=df1.withColumn("current_month", F.month(F.current_date())) # mois
courant
# last_day()
df1=df1.withColumn("last_day_of_month", F.last_day(F.current_date())) #
dernier jour du mois à la date courante
df1.show(truncate=False)
df1.printSchema()
```

4.6.40 Traiter une colonne de format json string : les fonctions from_json() et to_json()

La fonction from_json() permet d'extraire une colonne de type Map() à partir d'une valeur json de type string. A l'inverse, la colonne to_json() convertit une valeur de type Map() en une valeur json de type string. Les exemples ci-dessous illustrent l'utilisation des deux fonctions.

4.6.40.1 Convertir une colonne de type json string en une colonne de type Map() : la fonction from_json()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data=[("001",'{"nom":"Sylvain","ville":"Paris","age":22}'),
      ("002",'{"nom":"Laurent","ville":"New York","age":25}'),
      ("003",'{"nom":"Alice","ville":"Londres","age":27}')]

columns=["id","infos"]
df=spark.createDataFrame(data,columns)
df.show(truncate=False)
df.printSchema()

# convertir le json string en map()
df1=df.withColumn("infos_map",F.from_json(F.col("infos"),MapType(StringType
(),StringType())))
df1.drop(("infos").show(truncate=False)
df1.drop(("infos").printSchema()
```

```
# Output show() df
+---+----------------------------------------------+
|id |infos                                         |
+---+----------------------------------------------+
|001|{"nom":"Sylvain","ville":"Paris","age":22}    |
|002|{"nom":"Laurent","ville":"New York","age":25}|
|003|{"nom":"Alice","ville":"Londres","age":27}    |
+---+----------------------------------------------+
# output printSchema df
root
 |-- id: string (nullable = true)
 |-- infos: string (nullable = true)
# Output show() df1
+---+-----------------------------------------------+
|id |infos_map                                      |
+---+-----------------------------------------------+
|001|[nom -> Sylvain, ville -> Paris, age -> 22]    |
```

```
|002|[nom -> Laurent, ville -> New York, age -> 25]|
|003|[nom -> Alice, ville -> Londres, age -> 27]    |
+---+----------------------------------------------+
# output printSchema df1
root
 |-- id: string (nullable = true)
 |-- infos_map: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)
```

4.6.40.2 Convertir une colonne de type Map() en une colonne de type json string : la fonction to_json()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

# Créer d'abord une colonne map
data=[("001",'{"nom":"Sylvain","ville":"Paris","age":22}'),
      ("002",'{"nom":"Laurent","ville":"New York","age":25}'),
      ("003",'{"nom":"Alice","ville":"Londres","age":27}')]

columns=["id","infos_map"]
df=spark.createDataFrame(data,columns)
# convertir le json string en map()
df=df.withColumn("infos_map",F.from_json(F.col("infos_map"),MapType(StringT
ype(),StringType())))
df.show(truncate=False)
df.printSchema()

## Créer le json à partir du map:
df1=df.withColumn("infos_json",F.to_json(F.col("infos_map")))
df1.drop("infos_map").show(truncate=False)
df1.drop("infos_map").printSchema()
```

```
# Output show() df
+---+----------------------------------------------+
|id |infos_map                                     |
+---+----------------------------------------------+
|001|[nom -> Sylvain, ville -> Paris, age -> 22]   |
|002|[nom -> Laurent, ville -> New York, age -> 25]|
|003|[nom -> Alice, ville -> Londres, age -> 27]    |
+---+----------------------------------------------+
```

```
# output printSchema df
root
 |-- id: string (nullable = true)
 |-- infos map: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)

# Output show() df1
+---+-------------------------------------------------+
|id |infos_json                                       |
+---+-------------------------------------------------+
|001|{"nom":"Sylvain","ville":"Paris","age":"22"}     |
|002|{"nom":"Laurent","ville":"New York","age":"25"}|
|003|{"nom":"Alice","ville":"Londres","age":"27"}     |
+---+-------------------------------------------------+
# output printSchema df1
root
 |-- id: string (nullable = true)
 |-- infos_json: string (nullable = true)
```

4.6.41 Créer une vue Hive temporaire à partir d'un dataframe : la fonction createOrReplaceTempView()

De même qu'on peut exécuter une requête sur une table physique Hive via spark.sql(), on peut aussi exécuter une requête sur une vue hive temporaire créée à l'intérieur de la sessions Spark. Pour créer une vue Hive temporaire, on utilise la fonction createOrReplaceTempView(). La vue ainsi créée se comporte comme une table hive normale sur laquelle on peut exécuter une requête HQL standard via le module Spark.sql() L'exemple ci-dessous montre la création d'une table hive temporaire à partir d'un Dataframe et l'interrogation de cette vue via le module spark.sql().

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)
df.printSchema()
```

```
# Créer une vue Hive temporaire
df.createOrReplaceTempView("EMPLOYEES")

#### Appliquer spark.sql() sur la vue créée
df1 = spark.sql("SELECT * from EMPLOYEES")
df1.show(truncate=False)
df1.printSchema()

df2 = spark.sql("SELECT genre, count(*) as count from EMPLOYEES group by
genre order by genre")
df2.show(truncate=False)
df2.printSchema()
```

```
# Output show() df
+---+--------+------+----------+----------+-----+-------+
|id |prenom  |surnom|nom       |date_naiss|genre|salaire|
+---+--------+------+----------+----------+-----+-------+
|01 |Pascal  |      |Martin    |1995-02-25|M    |4500   |
|02 |Pierre  |Rose  |          |1998-07-12|M    |3000   |
|03 |Laurent |      |Bernard   |2000-09-14|M    |3000   |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|05 |Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+---+--------+------+----------+----------+-----+-------+
# output printSchema df
root
 |-- id: string (nullable = true)
 |-- prenom: string (nullable = true)
 |-- surnom: string (nullable = true)
 |-- nom: string (nullable = true)
 |-- date_naiss: string (nullable = true)
 |-- genre: string (nullable = true)
 |-- salaire: long (nullable = true)

# Output show() df1
+---+--------+------+----------+----------+-----+-------+
|id |prenom  |surnom|nom       |date_naiss|genre|salaire|
+---+--------+------+----------+----------+-----+-------+
|01 |Pascal  |      |Martin    |1995-02-25|M    |4500   |
|02 |Pierre  |Rose  |          |1998-07-12|M    |3000   |
|03 |Laurent |      |Bernard   |2000-09-14|M    |3000   |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|05 |Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+---+--------+------+----------+----------+-----+-------+
# output printSchema df1
root
 |-- id: string (nullable = true)
 |-- prenom: string (nullable = true)
 |-- surnom: string (nullable = true)
 |-- nom: string (nullable = true)
 |-- date_naiss: string (nullable = true)
 |-- genre: string (nullable = true)
 |-- salaire: long (nullable = true)

# Output show() df2

+-----+-----+
|genre|count|
+-----+-----+
|F    |2    |
```

```
|M    |3    |
+-----+-----+

# output printSchema df2
root
 |-- genre: string (nullable = true)
 |-- count: long (nullable = false)
```

4.6.42 Utiliser une expression SparkSQL : la fonction expr()

Une expression SparkSQL est une expression dans laquelle on utilise les syntaxes propres au langage HQL à la place des fonction natives Spark pour effectuer des opérations de traitement sur les colonnes d'un dataframe. Nous savons déjà que la plupart des fonctions natives Spark disponibles dans le package pyspark.sql.functions sont identiques aux fonctions standards du langage HQL. Par exemples les fonctions de traitement: substring(), regexp_replace(), current_date(), current_timestamp(), date_add(), etc.. Cette compatibilité entre les fonctions natives Spark avec le langage HQL fait qu'on peut directement utiliser une syntaxe de type HQL à la place d'une fonction native Spark lors d'une opération de transformation d'un dataframe. Toutefois, pour pouvoir utiliser une expression de type HQL sur un dataframe, cette expression doit être encapsuler dans une fonction Spark intermédiaire qui se charge de traduire la syntaxe HQL en syntaxe fonctionnelle Spark. Cette fonction d'intermédiation est la fonction expr().

L'exemple ci-dessous quelques utilisations de la fonction exp() pour encapsuler les expressions de type SparkSQL et de les appliquer aux colonnes d'un dataframe. En plus de la fonction withColum(), la fonction expr() peut être utilisée avec de nombreuses autres fonctions spark comme filter(), etc.. Voir exemple ci-dessous.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
```

```
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

### Utiliser la fonction expr() à l'intérieur d'un withColumn()
# Extraire l'année :
df1=df.withColumn("annee_naiss", F.expr("substring(date_naiss,0,4)"))
df1.show(truncate=False)

# Créer la colonne sexe
df2=df.withColumn("sexe", F.expr("CASE WHEN genre = 'M' THEN 'Homme' WHEN
genre = 'F' THEN 'Femme' ELSE null END"))
df2.show(truncate=False)

# Remplacer le caractère 0 par xx dans la colonne id
df3=df.withColumn("id", F.expr("regexp_replace(id,'0','xx')"))
df3.show(truncate=False)

# Générer calculer la date du prochain anniversaire de l'employe: ajouter
12 mois à la colonne date_naiss.
df4=df.withColumn("date_anniv", F.expr("add_months(CAST(date_naiss as
date),12)"))
df4.show(truncate=False)

### Utiliser la fonction expr() à l'intérieur d'un filter()

df5=df.filter(F.expr("substring(date_naiss,0,4)")=="1998")
df5.show()
```

```
# Output show() df
+---+--------+------+----------+----------+-----+-------+
|id |prenom  |surnom|nom       |date_naiss|genre|salaire|
+---+--------+------+----------+----------+-----+-------+
|01 |Pascal  |      |Martin    |1995-02-25|M    |4500   |
|02 |Pierre  |Rose  |          |1998-07-12|M    |3000   |
|03 |Laurent |      |Bernard   |2000-09-14|M    |3000   |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|05 |Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+---+--------+------+----------+----------+-----+-------+

# Output show() df1
+---+--------+------+----------+----------+-----+-------+-----------+
|id |prenom  |surnom|nom       |date naiss|genre|salaire|annee naiss|
+---+--------+------+----------+----------+-----+-------+-----------+
|01 |Pascal  |      |Martin    |1995-02-25|M    |4500   |1995       |
|02 |Pierre  |Rose  |          |1998-07-12|M    |3000   |1998       |
|03 |Laurent |      |Bernard   |2000-09-14|M    |3000   |2000       |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |1989       |
|05 |Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |1982       |
+---+--------+------+----------+----------+-----+-------+-----------+

# Output show() df2
+---+--------+------+----------+----------+-----+-------+-----+
|id |prenom  |surnom|nom       |date_naiss|genre|salaire|sexe |
+---+--------+------+----------+----------+-----+-------+-----+
|01 |Pascal  |      |Martin    |1995-02-25|M    |4500   |Homme|
|02 |Pierre  |Rose  |          |1998-07-12|M    |3000   |Homme|
|03 |Laurent |      |Bernard   |2000-09-14|M    |3000   |Homme|
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |Femme|
|05 |Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |Femme|
+---+--------+------+----------+----------+-----+-------+-----+
```

```
# Output show() df3
+---+--------+------+----------+----------+-----+-------+
|id |prenom  |surnom|nom       |date_naiss|genre|salaire|
+---+--------+------+----------+----------+-----+-------+
|xx1|Pascal  |      |Martin    |1995-02-25|M    |4500   |
|xx2|Pierre  |Rose  |          |1998-07-12|M    |3000   |
|xx3|Laurent |      |Bernard   |2000-09-14|M    |3000   |
|xx4|Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|xx5|Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+---+--------+------+----------+----------+-----+-------+

# Output show() df4

+---+--------+------+----------+----------+-----+-------+----------+
|id |prenom  |surnom|nom       |date_naiss|genre|salaire|date_anniv|
+---+--------+------+----------+----------+-----+-------+----------+
|01 |Pascal  |      |Martin    |1995-02-25|M    |4500   |1996-02-25|
|02 |Pierre  |Rose  |          |1998-07-12|M    |3000   |1999-07-12|
|03 |Laurent |      |Bernard   |2000-09-14|M    |3000   |2001-09-14|
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |1990-10-06|
|05 |Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |1983-02-15|
+---+--------+------+----------+----------+-----+-------+----------+

# Output show() df5
+---+------+------+---+----------+-----+-------+
| id|prenom|surnom|nom|date_naiss|genre|salaire|
+---+------+------+---+----------+-----+-------+
| 02|Pierre|  Rose|   |1998-07-12|    M|   3000|
+---+------+------+---+----------+-----+-------+
```

4.7 Actions sur un dataframe

4.7.1 Compter le nombre de lignes d'un Dataframe : la fonction count()

La fonction count() renvoie le nombre de lignes d'un dataframe

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
```

```
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)
df.printSchema()

# Comptage
print("Nb records:"+str(df.count()))
```

```
# Output show() df
+---+--------+------+----------+----------+-----+-------+
|id |prenom  |surnom|nom       |date_naiss|genre|salaire|
+---+--------+------+----------+----------+-----+-------+
|01 |Pascal  |      |Martin    |1995-02-25|M    |4500   |
|02 |Pierre  |Rose  |          |1998-07-12|M    |3000   |
|03 |Laurent |      |Bernard   |2000-09-14|M    |3000   |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|05 |Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+---+--------+------+----------+----------+-----+-------+
# output count df
Nb records:5
```

4.7.2 Afficher un nombre défini de ligne d'un Dataframe : la fonction show()

Cette fonction affiche le contenu d'un dataframe. Par défaut, la fonction affiche les 20 premières lignes du dataframe ainsi que les 20 premiers caractères de la valeur de chaque colonne. Mais on peut modifier ce comportement par défaut en renseignant l'argument n et l'argument truncate. Par exemple, on peut choisir n=100 pour afficher les 100 premières lignes du dataframe. Et on peut choisir truncate=False pour afficher la valeur entière d'une colonne sans troncature. Il est également possible de spécifier l'argument vertical=True lorsque l'on sait afficher le contenu du dataframe de manière vertical, c'est-à-dire pour chaque ligne, afficher les valeurs colonne par colonne dans le sens vertical. L'exemple ci-dessous illustre l'utilisation de la fonction show()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
```

```
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)

df.show() # par défaut affiche au max les 20 premières lignes du df.
df.show(n=100) # Spécifie le nombre de lignes à afficher. ici 100
df.show(truncate=False) # Affiche le contenu des colonnes sans tronquer les
valeurs affichées
df.show(vertical=True) # Affiche le contenu du df en vertical
df.show(n=20,truncate=False,vertical=True) # Affiche le contenu du df en
vertical sans tronquer les valeurs affichées
```

```
# Output show() df vertical=False
+---+--------+------+----------+----------+-----+-------+
| id|  prenom|surnom|       nom|date_naiss|genre|salaire|
+---+--------+------+----------+----------+-----+-------+
| 01|  Pascal|      |    Martin|1995-02-25|    M|   4500|
| 02|  Pierre|  Rose|          |1998-07-12|    M|   3000|
| 03| Laurent|      |   Bernard|2000-09-14|    M|   3000|
| 04|Virginie|  Anne|Carpentier|1989-10-06|    F|   2500|
| 05|   Jenny|Carole|   Bouvier|1982-02-15|    F|   3500|
+---+--------+------+----------+----------+-----+-------+

# Output show() df vertical=True
-RECORD 0----------------
 id         | 01
 prenom     | Pascal
 surnom     |
 nom        | Martin
 date_naiss | 1995-02-25
 genre      | M
 salaire    | 4500
-RECORD 1----------------
 id         | 02
 prenom     | Pierre
 surnom     | Rose
 nom        |
 date_naiss | 1998-07-12
 genre      | M
 salaire    | 3000
-RECORD 2----------------
 id         | 03
 prenom     | Laurent
 surnom     |
 nom        | Bernard
 date_naiss | 2000-09-14
 genre      | M
 salaire    | 3000
-RECORD 3----------------
 id         | 04
 prenom     | Virginie
 surnom     | Anne
 nom        | Carpentier
 date_naiss | 1989-10-06
 genre      | F
```

```
 salaire    | 2500
-RECORD 4----------------
 id         | 05
 prenom     | Jenny
 surnom     | Carole
 nom        | Bouvier
 date_naiss | 1982-02-15
 genre      | F
 salaire    | 3500
```

4.7.3 Récupérer le contenu d'un Dataframe : la fonction collect()

La fonction collect() permet de convertir le dataframe en une array d'objets de type de Row et récupère l'ensemble sur le driver. La fonction collect() est souvent utilisée lorsque l'on souhaite passer d'un mode de traitement parallélisé du dataframe à une collection standard de données sur laquelle on applique les fonctions de traitement du package standard python. L'utilisation de la fonction collect() doit être faite avec beaucoup de précaution car lorsque le dataframe collecté ne peut tenir sur la mémoire du driver Spark, le traitement tombe systématiquement en erreur. L'exemple ci-dessous illustre l'utilisation de la fonction collect().

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

# Récup du dataframe
elems = df.collect() #
print("elems:"+ str(elems)) # Afficher chaque ligne: objet Row

# Boucle sur les objets row:

#Méthod1 : afficher à partir du nom de la colonne
for row in elems:
    print( row['id'] + ","+row['prenom'] + ","+ row['surnom'] + ","
+row['nom'] + "," +row['date_naiss'] + "," +str(row['salaire']))
```

```
#Méthode 2 : afficher à partir de l'index de la colonne
for i in range(len(elems)):
    print(elems[i][0] + "," +elems[i][1] + "," +elems[i][2] + ","
+elems[i][3] + "," +str(elems[i][4]))
```

```
# Output show() df
+---+--------+------+----------+----------+-----+-------+
|id |prenom  |surnom|nom       |date_naiss|genre|salaire|
+---+--------+------+----------+----------+-----+-------+
|01 |Pascal  |      |Martin    |1995-02-25|M    |4500   |
|02 |Pierre  |Rose  |          |1998-07-12|M    |3000   |
|03 |Laurent |      |Bernard   |2000-09-14|M    |3000   |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|05 |Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+---+--------+------+----------+----------+-----+-------+

# Output collect() print
elems:[Row(id='01', prenom='Pascal', surnom='', nom='Martin',
date_naiss='1995-02-25', genre='M', salaire=4500), Row(id='02',
prenom='Pierre', surnom='Rose', nom='', date_naiss='1998-07-12', genre='M',
salaire=3000), Row(id='03', prenom='Laurent', surnom='', nom='Bernard',
date_naiss='2000-09-14', genre='M', salaire=3000), Row(id='04',
prenom='Virginie', surnom='Anne', nom='Carpentier', date_naiss='1989-10-
06', genre='F', salaire=2500), Row(id='05', prenom='Jenny',
surnom='Carole', nom='Bouvier', date_naiss='1982-02-15', genre='F',
salaire=3500)]

# Output print boucle méthode 1

01,Pascal,,Martin,1995-02-25,4500
02,Pierre,Rose,,1998-07-12,3000
03,Laurent,,Bernard,2000-09-14,3000
04,Virginie,Anne,Carpentier,1989-10-06,2500
05,Jenny,Carole,Bouvier,1982-02-15,3500

# Output print boucle méthode 2

01,Pascal,,Martin,1995-02-25
02,Pierre,Rose,,1998-07-12
03,Laurent,,Bernard,2000-09-14
04,Virginie,Anne,Carpentier,1989-10-06
05,Jenny,Carole,Bouvier,1982-02-15
```

4.7.4 Ecrire/exporter un dataframe dans un fichier csv : la fonction write.csv()

La fonction write.csv() permet d'exporter le contenu d'un dataframe dans un fichier csv. L'exemple ci-dessous montre l'utilisation de la fonction write.csv() pour stocker un fichier sur hdfs avec différentes options d'écriture : header, partionnement, etc.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
```

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

# Ecrire avec mode overwrite sans partitionnement
df.write.option("header",True)\
            .option("delimiter", ";")\
            .mode("overwrite") \
            .csv("hdfs://nnde01/user/mk/csv_data/output/nopart1")

# Ecrire avec mode overwrite avec partitionnement
df.write.partitionBy("genre")\
            .option("header",True)\
            .option("delimiter", ";")\
            .mode("overwrite") \
                  .csv("hdfs://nnde01/user/mk/csv_data/output/part1")

# Ecrire avec mode append sans partitionnement
df.write.option("header",True)\
            .option("delimiter", ";")\
            .mode("append") \
            .csv("hdfs://nnde01/user/mk/csv_data/output/nopart2")

# Ecrire avec mode append avec partitionnement
df.write.partitionBy("genre")\
            .option("header",True)\
            .option("delimiter", ";")\
            .mode("append") \
                  .csv("hdfs://nnde01/user/mk/csv_data/output/part2")
```

4.7.5 Ecrire/exporter un dataframe dans un fichier texte plat : la fonction write.text()

La fonction write.text() permet d'écrire le contenu d'un dataframe dans un fichier texte plat. L'exemple ci-dessous montre l'utilisation de la fonction write.txt() pour écrire le contenu d'un dataframe dans un fichier texte stocké sur hdfs.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
# Rassembler toutes les valeurs dans une seule colonne avant export

df.show(truncate=False)

# Ecrire avec mode overwrite sans partitionnement
df1 =df.select(F.concat_ws(";", *columns).alias("combined_cols") ) # concat
avec séparateur ;
df1.coalesce(1).write.option("header",True)\
            .mode("overwrite") \
            .text("hdfs://nnde01/user/mk/txt_data/output/nopart1")

# Ecrire avec mode overwrite avec partitionnement
selected_cols=df.columns
selected_cols.remove("genre")
df2
=df.select("genre",F.concat_ws(";",*selected_cols).alias("combined_cols") )
# concat avec séparateur ;
df2.coalesce(1).write.partitionBy("genre")\
            .option("header",True)\
            .mode("overwrite") \
                  .text("hdfs://nnde01/user/mk/txt_data/output/part1")

# Ecrire avec mode append sans partitionnement
df3 =df.select(F.concat_ws(";", *columns).alias("combined_cols") )
df3.coalesce(1).write.option("header",True)\
            .mode("append") \
            .text("hdfs://nnde01/user/mk/txt_data/output/nopart2")

# Ecrire avec mode append avec partitionnement
selected_cols=df.columns
selected_cols.remove("genre")
df4
=df.select("genre",F.concat_ws(";",*selected_cols).alias("combined_cols") )
# concat avec séparateur ;
df4.coalesce(1).write.partitionBy("genre")\
            .option("header",True)\
            .mode("append") \
                  .text("hdfs://nnde01/user/mk/txt_data/output/part2")
```

4.7.6 Ecrire/exporter un dataframe dans un fichier texte json : la fonction write.json()

L'exemple ci-dessous montre l'utilisation de la fonction write.json() pour écrire le contenu du dataframe dans un fichier json plat. L'exemple ci-dessous montre l'utilisation de la fonction write.json() pour écrire le contenu d'un dataframe dans un fichier json stocké sur hdfs.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
# Rassembler toutes les valeurs dans une seule colonne avant export

df.show(truncate=False)

# Ecrire avec mode overwrite
df.coalesce(1).write.option("header",True)\
            .mode("overwrite") \
            .json("hdfs://nnde01/user/mk/json_data/output/nopart1")

# Ecrire avec mode append
df.coalesce(1).write.option("header",True)\
            .mode("append") \
            .json("hdfs://nnde01/user/mk/json_data/output/nopart2")
```

4.7.7 Ecrire/exporter un dataframe dans un fichier parquet : la fonction write.parquet()

La fonction write.parquet() permet d'écrire le contenu d'un dataframe dans un fichier parquet. L'exemple ci-dessous montre l'utilisation de la fonction write.parquet() pour écrire le contenu d'un dataframe stocké sur hdfs.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json


spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
# Rassembler toutes les valeurs dans une seule colonne avant export

df.show(truncate=False)

# mode overwrite sans partitionnement
df.write.mode('overwrite')\
      .parquet("hdfs://nnde01/user/mk/parquet_data/output/nopart1")

# mode overwrite avec partitionnement
df.write.partitionBy("genre")\
            .mode('overwrite')\
            .parquet("hdfs://nnde01/user/mk/parquet_data/output/part1")

# mode append sans partitionnement
df.write.mode('append')\
      .parquet("hdfs://nnde01/user/mk/parquet_data/output/nopart2")

# mode append avec partitionnement
df.write. partitionBy("genre")\
      .mode('append')\
            .parquet("hdfs://nnde01/user/mk/parquet_data/output/part2")
```

4.7.8 Ecrire/exporter un dataframe dans un fichier de format orc : la fonction write().format("orc").save()

La fonction write() utilisée avec l'option format("orc") permet d'écrire le contenu d'un dataframe dans un fichier de format orc. La fonction est completée par l'utilisation de la fonction save() qui permet d'indiquer le répertoire de stockage du fichier. L'exemple ci-dessous montre l'utilisation de la fonction write().format("orc") pour écrire le contenu dans un fichier sur hdfs.

```
# Import de toutes les librairies utilitaires usuelles
```

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
# Rassembler toutes les valeurs dans une seule colonne avant export

df.show(truncate=False)

#mode overwrite sans partitionnement
df.write.mode('overwrite')\
      .format("orc")\
            .save("hdfs://nnde01/user/mk/orc_data/output/nopart1")

# mode overwrite avec partitionnement
df.write.partitionBy("genre")\
      .mode('overwrite')\
            .format("orc")\
                  .save("hdfs://nnde01/user/mk/orc_data/output/part1")

# mode append sans partitionnement
df.write.mode('append')\
      .format("orc").save("hdfs://nnde01/user/mk/orc_data/output/nopart2")

# mode append avec partitionnement
df.write.partitionBy("genre")\
      .mode('append')\
            .format("orc")\
                  .save("hdfs://nnde01/user/mk/orc_data/output/part2")
```

4.7.9 Ecrire/exporter un dataframe dans une table Hive : la fonction saveAstable()

La fonction saveAstable() permet d'écrire le contenu d'un dataframe dans une table Hive en mode overwrite ou en mode append. Les exemples ci-dessous montrent l'utilisation de la fonction saveAsTable() avec différentes options d'écriture.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
```

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
# Rassembler toutes les valeurs dans une seule colonne avant export

df.show(truncate=False)

###################  mode overwrite sans partitionnement
#########################
df.write.mode("overwrite")\
                  .option("translated_to_external", "true")\
                  .saveAsTable("mydatabase.mytable1"
                        ,path=
"hdfs_path_to_mydatabase/mydatabase.db/mytable1"
                        ,format="orc"
                        ,compression='snappy'
                        ,partitionBy=None )
# Reconfigurer la table si nécessaire
spark.sql("ALTER TABLE mydatabase.mytable1 SET TBLPROPERTIES
('discover.partitions'='TRUE', 'external.table.purge'='TRUE')")

# vérification de la table hive
spark.sql("select * from mydatabase.mytable1").show(truncate=False)

################### mode overwrite avec partitionnement ###################
df.write.mode("overwrite")\
            .option("translated_to_external", "true")\
            .saveAsTable("mydatabase.mytable2"
                  ,path= "hdfs_path_to_mydatabase/mydatabase.db/mytable2"
                  ,format="orc"
                  ,compression='snappy'
                  ,partitionBy=["genre"] )
# Reconfigurer la table et Resynchroniser avec hdfs si nécessaire
spark.sql("ALTER TABLE mydatabase.mytable2 SET TBLPROPERTIES
('discover.partitions'='TRUE', 'external.table.purge'='TRUE')")
spark.sql("MSCK REPAIR TABLEmydatabase.mytable2")

# # vérification de la table hive
spark.sql("select * from mydatabase.mytable2").show(truncate=False)


###################   En mode append sans partitionnement
###################

####################  Méthode 1: utiliser saveAsTable()
df.write.mode("append")\
```

```
                  .option("translated_to_external", "true")\
                  .saveAsTable("mydatabase.mytable1"
                        ,path=
"hdfs path to mydatabase/mydatabase.db/mytable1"
                        ,format="orc"
                        ,compression='snappy'
                        ,partitionBy=None )
# Reconfigurer la table si nécessaire
spark.sql("ALTER TABLE mydatabase.mytable1 SET TBLPROPERTIES
('discover.partitions'='TRUE', 'external.table.purge'='TRUE')")
#
################### Méthode 2 : Utiliser save(). A utiliser uniquement
lorsque la table existe déjà
df.write.mode("append")\
            .option("translated_to_external", "true")\
            .save(path= "hdfs_path_to_mydatabase/mydatabase.db/mytable1"
            ,format="orc"
            ,compression='snappy'
            ,partitionBy=None )
spark.sql("ALTER TABLE mydatabase.mytable1 SET TBLPROPERTIES
('discover.partitions'='TRUE', 'external.table.purge'='TRUE')")

# vérification de la table hive
spark.sql("select * from mydatabase.mytable1").show(truncate=False)

################### Mode append avec partitionnement   ###################

################### Méthode 1 : utiliser saveAsTable()

df.write.mode("append")\
                  .option("translated_to_external", "true")\
                  .saveAsTable("mydatabase.mytable2"
                        ,path=
"hdfs_path_to_mydatabase/mydatabase.db/mytable2"
                        ,format="orc"
                        ,compression='snappy'
                        ,partitionBy=["genre"] )
# Reconfigurer la table et Resynchroniser avec hdfs si nécessaire
spark.sql("ALTER TABLE mydatabase.mytable2 SET TBLPROPERTIES
('discover.partitions'='TRUE', 'external.table.purge'='TRUE')")
spark.sql("MSCK REPAIR TAB LEmydatabase.mytable2")

# vérification de la table hive
spark.sql("select * from mydatabase.mytable1").show(truncate=False)

################### Méthode 2 : utiliser save(). A utiliser uniquement
lorsque la table existe déjà
#spark.sql("drop tablemydatabase.mytable2")
df.write.mode("append")\
            .option("translated_to_external", "true")\
            .save(path= "hdfs_path_to_mydatabase/mydatabase.db/mytable2"
                  ,format="orc"
                  ,compression='snappy'
                  ,partitionBy=["genre"] )
spark.sql("ALTER TABLE mydatabase.mytable2 SET TBLPROPERTIES
('discover.partitions'='TRUE', 'external.table.purge'='TRUE')")
spark.sql("MSCK REPAIR TABLE mydatabase.mytable2")

# vérification de la table hive
spark.sql("select * from mydatabase.mytable2").show(truncate=False)
```

4.7.10 Convertir le dataframe PySpark en dataframe python Pandas : la fonction toPandas()

La fonction toPandas() permet de convertir un dataframe PySpark en un dataframe pandas stocké sur le Driver Spark. L'exemple ci-dessous montre l'utilisation de la fonction toPandas(). Signalons tout de même que l'utilisation de la fonction toPandas() doit être faite avec beaucoup de précaution car lorsque le dataframe PySpark est tel qu'il ne peut pas tenir sur la mémoire du Driver Spark, l'appel de la fonction toPandas() sur ce dataframe génèrera une erreur d'exécution.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json
import pandas as pd

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
# Rassembler toutes les valeurs dans une seule colonne avant export

df.show(truncate=False)

# Converttir le dataframe spark en dataframe pandas : toPandas()
pandas_df = df.toPandas()
# print du contenu
#méthode 1
print(pandas_df)
#méthode 2
print(pandas_df.to_string())
# Faire une boucle sur chaque ligne du dataframe et exécuter une
opérations:
for index,row in pandas_df.iterrows():

print(row["id"]+","+row["prenom"]+","+row["nom"]+","+row["date_naiss"]+","+
row["genre"]+","+str(row["salaire"]))
```

```
# Output show() df
+---+--------+------+----------+----------+-----+-------+
|id |prenom  |surnom|nom       |date_naiss|genre|salaire|
+---+--------+------+----------+----------+-----+-------+
```

```
|01 |Pascal   |       |Martin     |1995-02-25|M     |4500    |
|02 |Pierre   |Rose   |           |1998-07-12|M     |3000    |
|03 |Laurent  |       |Bernard    |2000-09-14|M     |3000    |
|04 |Virginie|Anne    |Carpentier|1989-10-06|F     |2500    |
|05 |Jenny    |Carole|Bouvier    |1982-02-15|F     |3500    |
+---+--------+------+-----------+----------+------+--------+

# Output  print pandas df
   id    prenom  surnom         nom  date_naiss genre  salaire
0  01    Pascal              Martin  1995-02-25     M     4500
1  02    Pierre    Rose              1998-07-12     M     3000
2  03   Laurent             Bernard  2000-09-14     M     3000
3  04  Virginie    Anne  Carpentier  1989-10-06     F     2500
4  05     Jenny  Carole     Bouvier  1982-02-15     F     3500
   id    prenom  surnom         nom  date_naiss genre  salaire
0  01    Pascal              Martin  1995-02-25     M     4500
1  02    Pierre    Rose              1998-07-12     M     3000
2  03   Laurent             Bernard  2000-09-14     M     3000
3  04  Virginie    Anne  Carpentier  1989-10-06     F     2500
4  05     Jenny  Carole     Bouvier  1982-02-15     F     3500

# Output print boucle sur les rows du pandas df
01,Pascal,Martin,1995-02-25,M,4500
02,Pierre,,1998-07-12,M,3000
03,Laurent,Bernard,2000-09-14,M,3000
04,Virginie,Carpentier,1989-10-06,F,2500
05,Jenny,Bouvier,1982-02-15,F,3500
```

4.8 Optimiser le traitement d'un Dataframe : repartitionnement, mise en cache et utilisation de variables partagées

4.8.1 Repartitionner un dataframe en utilisant la fonction repartition() ou la fonction coalesce()

Tout comme pour le RDD, les fonctions repartition() et coalesce() permettent de repartitionner un Dataframe. La fonction repartition() peut être utilisée pour augmenter ou diminuer le nombre de partitions d'un Dataframe. Et la fonction coalesce() n'est utilisable que pour diminuer le nombre de partitions. Les exemples ci-dessous illustrent l'utilisation des deux fonctions sur un Dataframe.

4.8.1.1 Cas 1 : Augmenter ou diminuer le nombre de partitions d'un dataframe : la fonction repartition()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
```

```
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

print("number of partitions:"+str(df.rdd.getNumPartitions()))
#Repartitions n= 4
df1=df.repartition(4)
print("number of partitions:"+str(df1.rdd.getNumPartitions()))

#Repartitions n= 7
df2=df.repartition(7)
print("number of partitions:"+str(df2.rdd.getNumPartitions()))
```

```
# Output show() df["value"]
+---+--------+------+----------+----------+-----+-------+
|id |prenom  |surnom|nom       |date_naiss|genre|salaire|
+---+--------+------+----------+----------+-----+-------+
|01 |Pascal  |      |Martin    |1995-02-25|M    |4500   |
|02 |Pierre  |Rose  |          |1998-07-12|M    |3000   |
|03 |Laurent |      |Bernard   |2000-09-14|M    |3000   |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|05 |Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+---+--------+------+----------+----------+-----+-------+
# Output print num partitions
number of partitions df:56
number of partitions df1:4
number of partitions df2:7
```

4.8.1.2 Cas 2 : Diminuer le nombre de partitions d'un dataframe : coalesce()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
```

```
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

print("number of partitions:"+str(df.rdd.getNumPartitions()))
#Repartitions n= 7
df1=df.coalesce(7)
print("number of partitions:"+str(df1.rdd.getNumPartitions()))

#Repartitions n= 4
df2=df.coalesce(4)
print("number of partitions:"+str(df2.rdd.getNumPartitions()))
```

```
# Output show() df["value"]
+---+--------+------+----------+----------+-----+-------+
|id |prenom  |surnom|nom       |date_naiss|genre|salaire|
+---+--------+------+----------+----------+-----+-------+
|01 |Pascal  |      |Martin    |1995-02-25|M    |4500   |
|02 |Pierre  |Rose  |          |1998-07-12|M    |3000   |
|03 |Laurent |      |Bernard   |2000-09-14|M    |3000   |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|05 |Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+---+--------+------+----------+----------+-----+-------+
# Output print num partitions
number of partitions df:56
number of partitions df1:7
number of partitions df2:4
```

4.8.2 Mise en cache et persistance d'un Dataframe : cache() et persist()

Les fonctions cache() et persist() sont des fonctions utilitaires permettant de stocker le Dataframe sur la mémoire de travail en vue d'accélérer les opérations de traitement. Les exemples ci-dessous montrent l'utilisation des fonctions cache() et persist().

4.8.2.1 Utilisation de la fonction cache()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json
```

```
spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

#Transformation 1
df1= df.withColumn("salaire",F.col("salaire")*1.10) #multiplier chaque
élément par 1.10
#Transformation 2
df2=df1.filter(F.col("salaire")> F.lit(4000)) #Retenir tous les salaires>
4000
#Mise en cache()
df2.cache()
#Action1
print("Nb records df2:"+str(df2.count()))
#Action2
elems = df2.collect()
print("elems:"+ str(elems))
for row in elems:
  print( row['id'] + ","+row['prenom'] + ","+ row['surnom'] + ","
+row['nom'] + "," +row['date_naiss'] + "," +str(row['salaire']))

df2.unpersist()
```

```
# Output show() df
+---+--------+------+----------+----------+-----+-------+
|id |prenom  |surnom|nom       |date_naiss|genre|salaire|
+---+--------+------+----------+----------+-----+-------+
|01 |Pascal  |      |Martin    |1995-02-25|M    |4500   |
|02 |Pierre  |Rose  |          |1998-07-12|M    |3000   |
|03 |Laurent |      |Bernard   |2000-09-14|M    |3000   |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|05 |Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+---+--------+------+----------+----------+-----+-------+
# Output print
Nb records df2:1
elems:[Row(id='01', prenom='Pascal', surnom='', nom='Martin',
date_naiss='1995-02-25', genre='M', salaire=4950.0)]
row printed: 01,Pascal,,Martin,1995-02-25,4950.0
```

4.8.2.2Utilisation de la fonction persist()

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
```

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [('01','Pascal','','Martin','1995-02-25','M',4500),
  ('02','Pierre','Rose','','1998-07-12','M',3000),
  ('03','Laurent','','Bernard','2000-09-14','M',3000),
  ('04','Virginie','Anne','Carpentier','1989-10-06','F',2500),
  ('05','Jenny','Carole','Bouvier','1982-02-15','F',3500)]

columns = ["id","prenom","surnom","nom","date_naiss","genre","salaire"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

#Transformation 1
df1= df.withColumn("salaire",F.col("salaire")*1.10) #multiplier chaque
élément par 1.10
#Transformation 2
df2=df1.filter(F.col("salaire")> F.lit(4000)) #Retenir tous les salaires>
4000
#Mise en cache()
df2.persist()
#Action1
print("Nb records df2:"+str(df2.count()))
#Action2
elems = df2.collect()
print("elems:"+ str(elems))
for row in elems:
  print( row['id'] + ","+row['prenom'] + ","+ row['surnom'] + ","
+row['nom'] + "," +row['date_naiss'] + "," +str(row['salaire']))

df2.unpersist()
```

```
# Output show() df
+---+--------+------+----------+----------+-----+-------+
|id |prenom  |surnom|nom       |date_naiss|genre|salaire|
+---+--------+------+----------+----------+-----+-------+
|01 |Pascal  |      |Martin    |1995-02-25|M    |4500   |
|02 |Pierre  |Rose  |          |1998-07-12|M    |3000   |
|03 |Laurent |      |Bernard   |2000-09-14|M    |3000   |
|04 |Virginie|Anne  |Carpentier|1989-10-06|F    |2500   |
|05 |Jenny   |Carole|Bouvier   |1982-02-15|F    |3500   |
+---+--------+------+----------+----------+-----+-------+
# Output print
Nb records df2:1
elems:[Row(id='01', prenom='Pascal', surnom='', nom='Martin',
date_naiss='1995-02-25', genre='M', salaire=4950.0)]
row printed: 01,Pascal,,Martin,1995-02-25,4950.0
```

La fonction unpersist() permet de dé-persister le dataframe dès que l'on a terminé les opérations de traitement. Elle peut être utilisée aussi bien à la suite de cache() ou persist().

4.8.2.3Niveau de mise en cache et de persistance d'un dataframe

Lorsqu'on utilise les fonctions cache() ou persist(), les résultats des calculs sont stockés par défaut sur la mémoire de travail. Cependant, ce comportement par défaut est insuffisant dans de nombreuses situations comme par exemple lorsque la taille du dataframe dépasse la taille de la mémoire. Mais Spark propose différentes options pour le stockage des résultats aussi bien sur la mémoire que sur le disque avec de multiples variantes. Les options présentées ci-dessous représentent les différents niveaux de stockage pouvant être utilisés en options des fonctions cache() et persist().

- **MEMORY_ONLY** : Stocke le dataframe sur la mémoire uniquement. C'est l'option par défaut de la fonction cache(). On peut aussi spécifier cache(MEMORY_ONLY).
- **MEMORY_ONLY_2** : Similaire à l'option MEMORY_ONLY mais réplique chaque partition sur deux executors distincts.
- **MEMORY_ONLY_SER** : stocke le dataframe uniquement sur la mémoire mais sous forme d'objets sérialisés. La sérialisation du dataframe permet de prendre moins d'espace que l'option MEMORY_ONLY.
- **MEMORY_ONLY_SER_2** : Similaire à l'option MEMORY_ONLY_SER mais réplique chaque partition sur deux executors distincts.
- **MEMORY_AND_DISK** : Stocke une partie du dataframe sur la mémoire JVM et l'excédent sur le disque lorsque la taille de la mémoire n'est pas suffisante pour contenir tout le dataframe.
- **MEMORY_AND_DISK_2** : Identique à l'option MEMORY_AND_DISK mais réplique chaque partition sur deux executors distincts.
- **MEMORY_AND_DISK_SER** : Similaire à l'option MEMORY_AND_DISK mais stocke le dataframe sous formes d'objets sérialisés.
- **MEMORY_AND_DISK_SER_2** : Similaire à l'option MEMORY_AND_DISK_SER mais réplique chaque partition sur deux executors distincts.
- **DISK_ONLY** : Stocke le dataframe uniquement sur le disque. Cette option nécessite beaucoup plus d'opération E/S lors du traitement du dataframe.
- **DISK_ONLY_2** : Similaire à l'option DISK_ONLY mais réplique chaque partition sur deux executors distincts.

4.8.3 Utiliser des variables partagées lors du traitement d'un dataframe : les fonctions broadcast() et accumulator()

Tout comme pour les RDDs, deux variables partagées sont couramment utilisées dans les traitements des dataframes, à savoir les variables broadcastées et les variables accumulées.

Pour rappel, les variables broadcastées sont des variables envoyées à chaque executor afin faciliter les opérations de traitement des partitions du dataframe situés sur cet executor. Les variables accumulées sont des variables permettant d'obtenir des informations agrégées sur le dataframe sans avoir besoin d'appliquer une action : ex : comptage du nombre d'éléments d'un dataframe, somme d'une colonne, etc.

Les exemples ci-dessous illustrent les cas d'utilisation de la fonction broadcast() et de la fonction accumulator().

4.8.3.1 Cas de la fonction broadcast()

Le broadcast consiste, par exemple, à déposer une copie d'un dataset dans chaque executor près des partitions du dataframe à traiter. Le dépôt de la copie du dataset dans l'executor assure la data locality et évite les multiples transferts de données via le réseau (shuffle) qui sont parfois très couteux. Dans l'exemple ci-dessous, nous déposons une copie de la collection data sur chaque executor du Job Spark.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()

data = [(45,),(30,),(30,),(25,),(35,)]
# N'importe quelle type de données peut être broadcasté: df, collection,
etc..
bc = spark.sparkContext.broadcast(data)
print(bc.value)
```

4.8.3.2Cas de la fonction accumulator()

Une variable accumulator renvoie une information agrégée sur un dataframe. Les exemples ci-dessus illustrent l'utilisation de la fonction accumulator() pour calculer le nombre de lignes et la somme des valeurs d'une colonne d'un dataframe.

```
# Import de toutes les librairies utilitaires usuelles
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as F
from pyspark.sql.window import Window
import json

spark =
SparkSession.builder.master("yarn").appName("mysparkjob").enableHiveSupport
().getOrCreate()
```

```
data = [(45,),(30,),(30,),(25,),(35,)]

columns = ["score"]
df = spark.createDataFrame(data=data, schema = columns)
df.show(truncate=False)

#Transformer d'abord le dataframe en rdd avant d'appeler l'accumulator
rdd=df.rdd
#Accumulator de comptage des éléments du rdd
ac1=spark.sparkContext.accumulator(0)
rdd.foreach(lambda x:ac1.add(1))
print(ac1.value) #renvoie la valeur finale accumulée.

#Accumulator de somme des éléments du rdd : ici sur la première colonne
ac2= spark.sparkContext.accumulator(0)
rdd.foreach(lambda x:ac2.add(x[0]))
print(ac2.value) #renvoie la valeur finale accumulée (somme des valeurs de
la colonne initiale salaire.
```

```
# Output show() df
+-----+
|score|
+-----+
|45   |
|30   |
|30   |
|25   |
|35   |
+-----+
# Output print value accumulator
ac1:5
ac2:165
```

5 LANCER LE JOB PYSPARK : PREPARER L'ENVIRONNEMENT VIRTUEL PYTHON ET PARAMETRER LA COMMANDE SHELL SPARK-SUBMIT

Il existe deux prérequis à l'exécution d'un job pyspark sur le cluster. D'une part, il doit exister un environnement virtuel python contenant toutes les librairies python utilisés lors du codage du job. Ex : pandas, numpy, les librairies natives python et autres librairies externes. A noter qu'il n'est pas nécessaire d'installer la librairie package dans l'environnement virtuel car le package PySpark est fourni par le cluster lui-même. L'exécution d'un job Spark, d'autre part, nécessite le paramétrage de la commande shell spark-submit qui permet de soumettre le job Spark au cluster manager en vue de son exécution. Les deux sections ci-dessous montrent comment préparer un environnement virtuel python et comment paramétrer la commande spark-submit.

5.1.1 Construire l'environnement virtuel python venv

Le script shell spécifié ci-dessous permet de créer un environnement virtuel python. En plus des librairies standards python, nous ajoutons quelques librairies externes telles que numpy, pandas, pyarrow ,etc. Voir le script si-dessous.

```
#!/bin/sh
# ceration des environement vrituelles
export VENV_PYTHON_PATH=/home/mokeita/spark_project/virtualenv # Répertoire
où le virtualenv sera stocké
export VIRTUAL_ENV_NAME=myspark_venv # Nom du virtualenv

cd  ${VENV_PYTHON_PATH}/

# Se positionner sur python et initialiser un venv
/usr/product/python/python-3.7.5/bin/python -m venv
${VENV_PYTHON_PATH}/${VIRTUAL_ENV_NAME}
deactivate
source ${VENV_PYTHON_PATH}/${VIRTUAL_ENV_NAME}/bin/activate

# Mettre à jour le pip
python3 -m pip install --upgrade pip

# Installer les librairies nécessaires
python3 -m pip install numpy
python3 -m pip install pandas
python3 -m pip install pyarrow
python3 -m pip install venv-pack

# Création de l'archive du virtual env
chmod -R 775  ${VENV_PYTHON_PATH}/${VIRTUAL_ENV_NAME}
rm -rf  ${VENV_PYTHON_PATH}/${VIRTUAL_ENV_NAME}.tar.gz
venv-pack -o  ${VENV_PYTHON_PATH}/${VIRTUAL_ENV_NAME}.tar.gz # Générer
l'archive du virtualenv
chmod -R 775  ${VENV_PYTHON_PATH}/${VIRTUAL_ENV_NAME}.tar.gz
```

5.1.2 Paramétrer et lancer la commande shell spark-submit

La commande shell spark-submit permet de lancer le job spark. La commande ci-dessous montre un exemple de lancement du job PySpark en mode cluster managé par yarn.

```
spark-submit\
--name 'mysparkjob' \
--conf 'spark.yarn.appMasterEnv.PYSPARK_PYTHON=./myspark_venv/bin/python' \
--conf 'spark.yarn.maxAppAttempts=1' \
--conf 'spark.yarn.appMasterEnv.KRB5CCNAME=./tmp/mykeytab.50608' \
--conf 'spark.hadoop.dfs.nameservices=nnde01' \
--conf 'spark.hadoop.dfs.namenode.kerberos.principal.pattern=*' \
--conf 'spark.datasource.hive.warehouse.read.mode=DIRECT_READER_V2' \
--conf 'spark.sql.sources.partitionOverwriteMode=dynamic' \
--conf
'spark.sql.sources.commitProtocolClass=org.apache.spark.sql.execution.datas
ources.SQLHadoopMapReduceCommitProtocol' \
--master yarn \
--deploy-mode cluster \
--archives
/home/mokeita/spark_project/virtualenv/myspark_venv.tar.gz#myspark_venv \
--py-files
/home/mokeita/spark_project/bin/*,/tmp/mykeytab.50608#./tmp/mykeytab.50608
\
--files
/home/mokeita/spark_project/conf/*,/etc/hive/conf.cloudera.hive/hive-
site.xml \
--packages com.databricks:spark-avro_2.11:4.0.0,com.databricks:spark-
xml_2.11:0.4.1,org.apache.spark:spark-sql-kafka-0-10_2.11:2.3.0 \
--driver-memory 10g \
--num-executors 5 \
--executor-memory 8g \
--executor-cores 5 \
--queue test_queue \
/home/mokeita/spark_project/bin/main.py
```

Le lancement de la commande spark-submit nécessite un certain nombre de paramètres et d'options. Dans l'exemple ci-dessus les options les plus utilisées sont :

--name : qui permet d'attribuer un nom au job spark lancé.

--master : permet d'indiquer le type du cluster manager ou son adresse physique. Ici nous indiquons le cluster manager **yarn**. Mais dans d'autres types de cluster Spark comme mesos ou un cluster standalone, on indique l'adresse physique du master Spark. Dans des cas de tests Spark en local (hors cluster), on peut indiquer **local**. Cette valeur signifie que l'ensemble du job Spark s'exécutera en local sur la machine de l'utilisateur.

--deploy-mode : permet d'indiquer le mode de déploiement du Driver Spark. Ici nous utilisons le mode **cluster** qui signifie que le Driver Spark sera instancié sur un nœud du cluster, c'est-à-dire une des machines réquisitionnées par le cluster manager pour

exécuter l'application Spark. On peut aussi utiliser le deploy-mode **client** qui signifie que le Driver Spark sera instancié en local sur la machine d'où la commande spark-submit a été lancée. Il faut noter que cela soit en mode **cluster** ou en mode **client**, les executors Spark sont toujours lancés sur les nœuds du cluster : yarn, mesos, ou standalone Spark. Par contre, lorsque le master est de type **local**, alors le Driver et les executors sont situés sur la machine de l'utilisateur qui a lancé la commande spark-submit.

--driver-memory : permet la taille de la mémoire à attribuer au Driver Spark.

--num-executors : Permet d'indiquer le nombre d'executors Spark à utiliser.

--executor-memory : Permet d'indiquer la taille de mémoire à attribuer à chaque executor.

--executor-cores : permet d'indiquer le nombre de cores à créer dans chaque executor

--queue : Permet d'indiquer yarn à utiliser, ici le cluster manager est yarn.

--conf: qui permet d'ajouter une configuration au job spark. Les différentes configurations Spark utilisables sont disponibles à ce lien : https://spark.apache.org/docs/2.4.0/configuration.html

NB : Noter que les confs spécifiées dans le spark-submit sont moins prioriaires par rapport aux confs ajoutées à l'objet SparkSession initialisé dans le code PySpark. En effet, lorsqu'une conf est ajoutée à l'objet SparkSession, la valeur de cette conf écrase toutes les valeurs de la même conf spécifiée par ailleurs (spark-submit, fichiers de conf par défaut du cluster, etc..). C'est pourquoi, les confs settées lors du spark-submit doivent être mis en cohérence avec celle settées lors de l'initialisation de l'objet SparkSession dans le code Spark.

--archives : permet d'ajouter des fichiers archives utiles lors de l'exécution du traitement spark. Il peut s'agir des archives de type zip, tar.gz, etc. Dans l'exemple présenté ci-dessus, l'archive ajoutées correspond à l'environnement virtuel python que nous avons généré dans la section précédente.

--py-files : cette option permet d'ajouter des fichiers de types python utiles à la bonne exécution du job Spark. Ces fichiers sont situés en local dans une arborescence située sur la machine où la commande Spark-submit a été lancée.

--files : Cette option est similaire à l'option py-files à la seule différence que py-files est prévue pour les fichiers de type python alors que *--files* permet de charger n'importe quel type de fichiers.

--packages : C'est l'option prévue pour charger des packages externes spécifiques et qui peuvent utiles lors de l'exécution du job. Voir exemple ci-dessus.

./main.py : Le fichier main.py est le fichier python qui constitue le point d'entrée du code pysPark développé par l'utilisateur. Ce fichier est spécifié comme un argument

obligatoire de la commande spark-submit et non une option de type. Il n'est donc pas accompagné des caractères --.

6 RESSOURCES DOCUMENTAIRES

Tout d'abord, je tiens ici à remercier tous les auteurs de documents techniques, d'articles, de blogs et de ressources numériques dont la consultation m'a été utile lors de la rédaction de ce document. Je remercie plus particulièrement le site sparkbyexamples.com ainsi que tous ses contributeurs. Je remercie également les auteurs de toutes les ressources documentaires dont références n'apparaissent pas ici.

Bibliographie

Chambers B. et Zaharia M., (2018), Spark: The Definite Guide: Big Data Processing Made Simple, O'Reilly Media, Inc

Frampton Mike, (2015), Mastering Apache Spark: Gain Expertise In Processing And Storing Data By Using Advanced Techniques With Apache Spark, Packt Publishing

Guller Mohammed, (2015), Big Data Analytics with Spark, A Practitioner's Guide to Using Spark for Large Scale Data Analysis,

Karau H et Warren R, (2017), High-Performance Spark: Best Practices for Scaling and Optimizing Apache Spark, O'Reilly Media, Inc.

Karau H., Konwinski A, Wendell P. et Zaharia M, (2015), Learning Spark: Lightning-Fast Big Data Analysis, O'Reilly Media, Inc.

Yadav Rishi, (2015), Spark Cookbook, Packt Publishing

Urls utiles

https://spark.apache.org/docs/latest/sql-ref-datatypes.html
https://spark.apache.org/docs/2.4.0/configuration.html
https://sparkbyexamples.com/
https://spark.apache.org/docs/latest/api/python/reference/
https://spark.apache.org/documentation.html