Spark Documentation

Scala Documentation

Plan

1. Prendre en main le notebook
2. Prendre en main Scala
3. Découverte des APIS haut niveau
4. Manipulation de données plus avancée
5. Petit récapitulatif
6. UDF
7. Window function
8. Manipulation de données encore plus avancée

1. Prendre en main le notebook

Databricks met en place une plateforme pour tester Spark. Il y a quelques données dessus dans le système de fichiers distribué de Databricks.

Je vous propose de vous créer un compte. C'est là-dessus que nous ferons nos exercices.

Pour info Databricks a été fondé par les créateurs de Spark et agit en support de ce produit.

1. Se rendre sur cette URL https://databricks.com/try-databricks
2. Choisir la "community edition" en cliquant "Get started"
3. Remplir les champs
4. Cliquer "sign up"
5. Dans vos mails, cliquer sur le lien pour vérifier votre email
6. Choisissez votre mot de passe
7. Après avoir cliqué "Reset password", vous voilà dans la plateforme que nous allons utiliser
8. Cliquer sur "New notebook" pour ouvrir un nouveau notebook
9. Choisir le nom que vous voulez et bien spécifier "Scala"
10. Cliquer sur Detached
11. Cliquer sur "create a cluster"
12. Entrer un nom dans le champ "Cluster Name"
13. Cliquer "Create Cluster"
14. Retourner au notebook en cliquant "databricks" puis en choisissant votre notebook
15. Cliquer sur "Detached" et sélectionner le cluster récemment créé

2. Prendre en main Scala

Exercice 1

Créer une classe "Dog" pouvant représenter des chiens avec leur nom, âge et couleur.

Créer une fonction capable de prendre un chien en entrée et de renvoyer une string du type : "Le

chien Hachiko a 15 ans et il est de couleur marron".

Exercice 2

Créer une séquence de deux chiens. Afficher pour chacun une phrase du type : "Le chien Hachiko a 15 ans et il est de couleur marron".

3. Découverte des APIS haut niveau

Spark SQL : l'API ouverte à tous

Explications Spark SQL

Pour lire un fichier CSV avec Spark, on utilise cette fonction, le "show" nous permettant d'afficher les 20 premières lignes :

```
val simpleDiamonds = spark.read.csv("diamonds.csv")
simpleDiamonds.show
```

```
+----+-----+---------+-----+-------+-----+-----+-----+----+----+----+
| _c0|  _c1|      _c2|  _c3|    _c4|  _c5|  _c6|  _c7| _c8| _c9|_c10|
+----+-----+---------+-----+-------+-----+-----+-----+----+----+----+
|null|carat|      cut|color|clarity|depth|table|price|   x|   y|   z|
|   1| 0.23|    Ideal|    E|    SI2| 61.5|   55|  326|3.95|3.98|2.43|
|   2| 0.21|  Premium|    E|    SI1| 59.8|   61|  326|3.89|3.84|2.31|
|   3| 0.23|     Good|    E|    VS1| 56.9|   65|  327|4.05|4.07|2.31|
|   4| 0.29|  Premium|    I|    VS2| 62.4|   58|  334| 4.2|4.23|2.63|
|   5| 0.31|     Good|    J|    SI2| 63.3|   58|  335|4.34|4.35|2.75|
|   6| 0.24|Very Good|    J|   VVS2| 62.8|   57|  336|3.94|3.96|2.48|
|   7| 0.24|Very Good|    I|   VVS1| 62.3|   57|  336|3.95|3.98|2.47|
|   8| 0.26|Very Good|    H|    SI1| 61.9|   55|  337|4.07|4.11|2.53|
|   9| 0.22|     Fair|    E|    VS2| 65.1|   61|  337|3.87|3.78|2.49|
|  10| 0.23|Very Good|    H|    VS1| 59.4|   61|  338|   4|4.05|2.39|
|  11|  0.3|     Good|    J|    SI1|   64|   55|  339|4.25|4.28|2.73|
|  12| 0.23|    Ideal|    J|    VS1| 62.8|   56|  340|3.93| 3.9|2.46|
|  13| 0.22|  Premium|    F|    SI1| 60.4|   61|  342|3.88|3.84|2.33|
|  14| 0.31|    Ideal|    J|    SI2| 62.2|   54|  344|4.35|4.37|2.71|
|  15|  0.2|  Premium|    E|    SI2| 60.2|   62|  345|3.79|3.75|2.27|
|  16| 0.32|  Premium|    E|     I1| 60.9|   58|  345|4.38|4.42|2.68|
|  17|  0.3|    Ideal|    I|    SI2|   62|   54|  348|4.31|4.34|2.68|
|  18|  0.3|     Good|    J|    SI1| 63.4|   54|  351|4.23|4.29| 2.7|
|  19|  0.3|     Good|    J|    SI1| 63.8|   56|  351|4.23|4.26|2.71|
+----+-----+---------+-----+-------+-----+-----+-----+----+----+----+
```

Pour lire un fichier CSV avec Spark en prenant en compte le header, on utilise cette fonction :

```
val diamondsWithHeader = spark.read.option("header", "true").csv("diamonds.csv")
diamondsWithHeader.show
```

Pour imprimer le schéma, on utilise la fonction "printSchema" :

```
diamondsWithHeader.printSchema
```

```
root
 |-- _c0: string (nullable = true)
 |-- carat: string (nullable = true)
 |-- cut: string (nullable = true)
 |-- color: string (nullable = true)
 |-- clarity: string (nullable = true)
 |-- depth: string (nullable = true)
 |-- table: string (nullable = true)
 |-- price: string (nullable = true)
 |-- x: string (nullable = true)
 |-- y: string (nullable = true)
 |-- z: string (nullable = true)
```

Pour inférer le schéma de data dans un fichier csv :

```
val completeDiamonds = spark.read.option("header", "true").option("inferSchema",
"true").csv("diamonds.csv")
completeDiamonds.printSchema
```

```
root
 |-- _c0: integer (nullable = true)
 |-- carat: double (nullable = true)
 |-- cut: string (nullable = true)
 |-- color: string (nullable = true)
 |-- clarity: string (nullable = true)
 |-- depth: double (nullable = true)
 |-- table: double (nullable = true)
 |-- price: integer (nullable = true)
 |-- x: double (nullable = true)
 |-- y: double (nullable = true)
 |-- z: double (nullable = true)
```

Avec l'API "Spark SQL", nous allons requêter nos données. Pour cela, nous avons besoin de créer une table. On fait comme cela :

```
val people = spark.read.json("people.json")
people.createOrReplaceTempView("people")

val result = spark.sql("""
select * from people
""")

result.show
```

```
+----+-------+
| age|   name|
+----+-------+
|  12|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

On peut ensuite effectuer toute sorte de requêtes.

Par exemple, pour filtrer

```
val filteredPeople = spark.sql("""
select * from people where age > 20
""")

filteredPeople.show
```

```
+---+----+
|age|name|
+---+----+
| 30|Andy|
+---+----+
```

Pour sélectionner :

```
spark.sql("""
select name from people where age > 20
""").show
```

```
+----+
|name|
+----+
|Andy|
+----+
```

Pour réaliser quelques statistiques :

```
spark.sql("""
select max(age) as maxAge, min(age), round(avg(age))
from people
""").show
```

```
+--------+--------+-----------------+
|maxAge  |min(age)|round(avg(age), 0)|
```

```
+--------+--------+------------------+
|      30|      19|              25.0|
+--------+--------+------------------+
```

Beaucoup de choses disponibles en SQL sont disponibles avec Spark SQL : orderBy, limit, groupBy, etc.

Pour vous aider, vous pouvez utiliser des "show" et "printSchema" pour vérifier vos résultats.

Exercice 1

Data Diamonds : /databricks-datasets/Rdatasets/data-001/csv/ggplot2/diamonds.csv

Après avoir loadé les diamants et dans l'API Spark SQL, on veut :

Filtrer en excluant la couleur E

Exercice 2

Toujours avec les mêmes diamants, dans l'API Spark SQL, on veut :

Sélectionner uniquement les champs "cut", "clarity" et "depth"

Exercice 3

Toujours avec les mêmes diamants, dans l'API Spark SQL, on veut :

Calculer le prix minimum, le prix maximum, le prix moyen en arrondissant à l'unité après la virgule pour l'ensemble des diamants

Utiliser les alias "maxPrice", "minPrice", "avgPrice".

Pour rappel en SQL, un alias ressemble à ça : "as maxPrice".

Exercice 4

Toujours avec les mêmes diamants, dans l'API Spark SQL, on veut :

Calculer le prix minimum, le prix maximum, le prix moyen en arrondissant à l'unité après la virgule par couleur.

Utiliser les alias "maxPrice", "minPrice", "avgPrice".

Ordonner par "avgPrice".

Exercice 5

Toujours avec les mêmes diamants, dans l'API Spark SQL, on veut :

Calculer le prix minimum, le prix maximum, le prix moyen en arrondissant à l'unité après la virgule par carat.

Utiliser les alias "maxPrice", "minPrice", "avgPrice".

Ordonner par "avgPrice".

Que notez-vous par rapport au résultat de l'exercice 4 ?

Transformations et actions : le coeur de Spark

Explications transformations et actions

Dans Spark, il y a ce qu'on appelle des transformations qui, comme son nom l'indique, transforme la donnée en filtrant, ajoutant des éléments, sélectionnant certains champs,etc.

Et puis il y a des actions :

- Qui renvoie la donnée en console, comme "show"
- Qui renvoie dans l'API Scala, comme un "count" qui renvoie un "Int"
- Qui écrive la donnée vers l'extérieur (filesystem par exemple)

Exemples :

```
people.show
people.count
people.distinct.count //Dans ce cas-là c'est "count" qui fait l'action, "distinct"
est une transformation
people.write.csv("path")
```

Exercice 1

On veut savoir le nombre de diamants total

Exercice 2

On veut savoir le nombre distint de "cut"

DataFrame : une deuxième API haut niveau

DataFrame est plus developer friendly que Spark SQL et permet de faire les mêmes choses.

Pour lire un fichier, on fait exactement pareil que pour Spark SQL.

```
val diamonds = spark.read.csv("diamonds.csv")
```

Les "show" et "printSchema" existent aussi.

Spark SQL n'est qu'une manière de requêter la donnée. Les objets qu'elle manipule sont en fait des dataFrames.

```
import org.apache.spark.sql.DataFrame

val people: DataFrame = spark.read.json("people.json")
people.createOrReplaceTempView("people")

val result: DataFrame = spark.sql("""
select * from people
""")
```

On peut donc passer de l'un à l'autre.

Pour filtrer avec l'API DataFrame :

```
people.
  filter("age > 20").
  show
```

```
+---+----+
|age|name|
+---+----+
| 30|Andy|
+---+----+
```

Pour sélectionner avec l'API DataFrame :

```
people.
    select("name").
    filter("age > 20").
    show
```

```
+----+
|name|
+----+
|Andy|
+----+
```

Pour réaliser des statistiques :

```
import org.apache.spark.sql.functions._

people.
    agg(max("age").as("maxAge"), min("age"), round(avg("age"), 2)).
    show
```

```
+--------+--------+------------------+
|max(age)|min(age)|round(avg(age), 2)|
+--------+--------+------------------+
|      30|      19|              24.5|
+--------+--------+------------------+
```

Pour réaliser des statistiques par rapport à une colonne :

```
people.
    groupBy("name").
    agg(max("age"), min("age"), round(avg("age"))).
    show
```

Exercice 1

Après avoir loadé les diamants et dans l'API DataFrame, on veut :

Filtrer en excluant la couleur E

Exercice 2

Après avoir loadé les diamants et dans l'API DataFrame, on veut :

Sélectionner uniquement les champs "cut", "clarity" et "depth"

Exercice 3

Après avoir loadé les diamants et dans l'API DataFrame, on veut :

Calculer le prix minimum, le prix maximum, le prix moyen en arrondissant à l'unité pour l'ensemble des diamants

Utiliser les alias "maxPrice", "minPrice", "avgPrice".

Exercice 4

Après avoir loadé les diamants et dans l'API DataFrame, on veut :

Calculer le prix minimum, le prix maximum, le prix moyen en arrondissant à l'unité par couleur

Utiliser les alias "maxPrice", "minPrice", "avgPrice".

Ordonner par "avgPrice"

Dataset : une troisième API haut niveau

Explications Dataset

Les datasets sont une troisième API haut niveau. Elles apportent plus de sécurité parce qu'elles permettent de typer les dataFrames en les liant aux "case class".

L'inconvénient c'est qu'elles sont plus gourmandes.

Pour utiliser cette API, on part d'une DataFrame.

```
val people: DataFrame = spark.read.json("people.json")
```

On la lie à une "case class".

```
import org.apache.spark.sql.Dataset
case class People(age: Long, name: String)
val people: Dataset[People] =
spark.read.json("/FileStore/tables/people.json").as[People]
people.show
```

```
+----+-------+
| age|   name|
+----+-------+
|  12|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

Toutes les fonctions disponibles pour les dataFrames existent pour les datasets. Cependant, on a des techniques plus sécurisées pour manipuler ces objets.

Pour filtrer une dataset

```
val filteredPeople: Dataset[People] = people.filter(peoplePiece => {
  peoplePiece.age > 20
})

filteredPeople.show
```

Pour sélectionner un champ (attention là nous transformons le schéma de notre dataset) :

```
val selected: Dataset[Long] = people.map(peoplePiece => { peoplePiece.age })
selected.show
```

```
+-----+
|value|
```

```
+-----+
|   12|
|   30|
|   19|
+-----+
```

Pour effectuer des statistiques (attention là aussi nous transformons le schéma de notre dataset) :

```
val peopleStatistics = people.groupBy("name").agg(min("age").alias("minAge"),
max("age").alias("maxAge"), round(avg("age")).alias("avgAge"))

case class PeopleStatistics(name: String, minAge: Long, maxAge: Long, avgAge:
Double)
peopleStatistics.as[PeopleStatistics].show
```

```
+-------+------+------+------+
|   name|minAge|maxAge|avgAge|
+-------+------+------+------+
|Michael|    12|    12|  12.0|
|   Andy|    30|    30|  30.0|
| Justin|    19|    19|  19.0|
+-------+------+------+------+
```

Exercice 1

Aide : case class Diamond(_c0: Int, carat: Double, cut: String, color: String, clarity: String, depth: Double, table: Double, price: Int, x: Double, y: Double, z: Double)

Après avoir loadé les diamants et dans l'API Dataset, on veut :

Filtrer en excluant la couleur E

Exercice 2

Aide : case class Diamond(_c0: Int, carat: Double, cut: String, color: String, clarity: String, depth: Double, table: Double, price: Int, x: Double, y: Double, z: Double)

Après avoir loadé les diamants et dans l'API Dataset, on veut :

Sélectionner uniquement les champs "cut", "clarity" et "depth".

On peut là passer d'un type de dataset à un autre.

Exercice 3

Aide : case class Diamond(_c0: Int, carat: Double, cut: String, color: String, clarity: String, depth: Double, table: Double, price: Int, x: Double, y: Double, z: Double)

Après avoir loadé les diamants et dans l'API Dataset, on veut :

Calculer le prix minimum, le prix maximum, le prix moyen en arrondissant à l'unité pour l'ensemble des diamants

Exercice 4

Aide : case class Diamond(_c0: Int, carat: Double, cut: String, color: String, clarity: String, depth: Double, table: Double, price: Int, x: Double, y: Double, z: Double)

Après avoir loadé les diamants et dans l'API Dataset, on veut :

Calculer le prix minimum, le prix maximum, le prix moyen en arrondissant à l'unité par couleur et remapper la sortie dans une autre case class

4. Manipulation de données plus avancées

Créer des dataframes

Explications de création de dataFrames

Avec Spark, on peut créer soi-même ses dataFrames. C'est très pratique pour tester avec Spark.

```
val someDF = Seq(
  (8, "bat"),
  (64, "mouse"),
  (-27, "horse")
).toDF("number", "word")

someDF.show
someDF.printSchema
```

Le problème c'est que là on ne choisit pas son schéma.

On a la possibilité d'ajouter alors un schéma.

```
import org.apache.spark.sql._
import org.apache.spark.sql.types._

val someData = Seq(
  Row(8, "bat"),
  Row(64, "mouse"),
  Row(-27, "horse")
)

val someSchema = Seq(
  StructField("number", IntegerType, true),
  StructField("word", StringType, true)
)

val someDF = spark.createDataFrame(
```

```
  spark.sparkContext.parallelize(someData),
  StructType(someSchema)
)

someDF.show
someDF.printSchema
```

Exercice 1

Pour les données suivantes, créer une dataFrame sans schéma :

```
Batman, Super Héros, 32
Le Pingouin, Super Méchant, 45
Catwoman, Difficile à dire, 32
```

Exercice 2

Pour les mêmes données, créer une dataFrame avec schéma

Créer des datasets

Explications création de dataset

On peut aussi créer des datasets. Le plus simple est de partir des case class :

```
case class People(age: Long, name: String)
val people: Dataset[People] = Seq(
        People(12, "Lucien"),
        People(26, "Assia")
    ).toDS

people.show
```

On peut aussi créer une dataset vide :

```
val people: Dataset[People] = spark.emptyDataset[People]
people.show
```

Exercice 1

Créer une dataset pour les données suivantes (toujours les mêmes) :

```
Batman, Super Héros, 32
Le Pingouin, Super Méchant, 45
Catwoman, Difficile à dire, 32
```

Exercice 2

Créer une dataset vide à partir de la case class précédemment créée

Joindre les données

Explications

Avec Spark, on travaille souvent avec plusieurs dataFrames/datasets qu'on joint.

Avec Spark SQL, on ferait ainsi.

```
val people = spark.read.json("people.json")
val employees = spark.read.json("employees.json")

people.createOrReplaceTempView("people")
employees.createOrReplaceTempView("employees")

val result = spark.sql("""
select * from people
join employees on employees.name = people.name
""")

result.show
```

Avec l'API DataFrame/Dataset, on ferait ainsi :

```
val people = spark.read.json("people.json")
val employees = spark.read.json("employees.json")

val result = people.join(employees, employees.col("name") === people.col("name"),
"inner")

result.show
```

Exercice 1

Créer trois datasets pour correspondre aux case class suivantes :

```
case class Customer(id: Int, firstName: String)
case class Order(productId: Int, customerId: Int)
case class Product(id: Int, name: String, price: Float)
```

Les valeurs pour les datasets sont les suivantes : Customer :

```
1, "Sophie"
2, "Julien"
3, "Sarah"
```

```
4, "Irina"
5, "Renzo"
```

Order :

```
1, 3
2, 4,
4, 1
```

Product :

```
1, "Lego", 230.70F
2, "Dixit", 45.60F
3, "Batman figurine", 19.6F,
4, "Livre de coloriage", 3.5F
```

Exercice 2

A l'aide de ces datasets et dans l'API Spark SQL, on veut savoir qui a acheté un livre de coloriage (nom du customer, nom du produit)

Exercice 3

A l'aide de ces datasets et dans l'API DataFrame, on veut savoir qui a acheté un livre de coloriage (nom du customer, nom du produit)

Exercice 4

A l'aide de ces datasets et dans l'API Spark SQL, on veut savoir qui achète des produits valant plus de 200 euros (nom du customer, nom du produit, prix)

Exercice 5

A l'aide de ces datasets et dans l'API DataFrame, on veut savoir qui achète des produits valant plus de 200 euros (nom du customer, nom du produit, prix)

Ajouter une colonne

Explications ajout de colonne

On peut ajouter une colonne. Avec Spark SQL :

```
val people = spark.read.json("people.json")
people.createOrReplaceTempView("people")
val result = spark.sql("""
select 'something' as newColumn, name
from people
```

```
""")

result.show
```

Avec l'API DataFrame/Dataset :

```
import org.apache.spark.sql.functions.lit

val people = spark.read.json("people.json")
val result = people.withColumn("newColumn", lit("something"))
result.show
```

"lit" précise ici qu'on veut un littéral. On aurait pu utiliser un calcul.

```
def replaceLetters(col: Column): Column = {
  regexp_replace(col, "A", "B")
}

val people = spark.read.json("people.json")
val result = people.withColumn(
    "clean_words",
    replaceLetters(col("name"))
)

result.show
```

Exercice

Avec Spark SQL ou l'API DataFrame :

Ajouter aux données des diamants une colonne (du nom que vous voulez) pour avoir le cut en majuscule. Pour info, la fonction "upper" avec Spark est disponible via l'import "import org.apache.spark.sql.functions.upper" ou directement avec "upper" dans Spark SQL.

Renommer une colonne

Explications

Avec Spark, on peut aussi renommer une colonne soit via Spark SQL :

```
val people = spark.read.json("people.json")
people.createOrReplaceTempView("people")
val result = spark.sql("""
    select name as lastName
    FROM people
""")

result.show
```

```
```

Soit avec l'API DataFrame/Dataset :

```
val people = spark.read.json("people.json")
val result = people.withColumnRenamed("name", "lastName")

result.show
```

Exercice

Renommer une colonne de votre choix dans les données de diamants.

Supprimer une colonne

Explications

Avec Spark, on peut aussi supprimer une colonne. Soit en ne la nommant pas dans SparkSQL, soit via l'API DataFrame/Dataset :

```
val people = spark.read.json("people.json")
val result = people.drop("name")

result.show
```

Exercice

Supprimer une colonne de votre choix dans les données de diamants

5. Petit récapitulatif

Je vous propose une suite de petits exercices pour balayer tout ce que nous venons de voir.

Exercice 1

Quel est le nombre de couleurs différentes par type de coupe ("cut") ? Dans les fonctions Spark disponibles dans l'API DataFrame, il existe une fonction "countDistinct".

Exercice 2

Les données sont les suivantes :

```
Marie, thé vert, France
Andrea, tisane canelle, Italie
Yijia, thé oolong, Chine
Pi-Yuan, thé rose, Chine
Ao, thé litchi, Chine
```

```
Elena, thé noir, UK
Cory, thé earl grey, UK
```

Elles représentent dans l'ordre un nom de consommateur, un produit acheté et le pays du consommateur. A partir de celles-ci, il s'agit de trouver quel est le pays qui consomme le plus.

Exercice 3

Les données sont les suivantes. D'un côté nous avons des salariés avec leur nom, leur rôle et leur département :

```
Johnathan, Développeur, WEB
Lou, Développeur, WEB
Solène, PO, DATA
Selim, Développeur, DATA
Soraya, Développeur, WEB
```

De l'autre côté, nous avons les départements avec le nom et la description :

```
WEB, S'occupe du site web coorporate
DATA, S'occupe des grosses données
```

Nous voulons joindre aux premières données la description du département et passer le nom du département en minuscule.

Pour effectuer des calculs ou opérations sur des sélections :

```
// Select everybody, but increment the age by 1
df.select($"name", $"age" + 1).show()
// +-------+---------+
// |   name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |   Andy|       31|
// | Justin|       20|
// +-------+---------+
```

Exercice 4 : Cas de calcul de résultats d'AB Testing

Supposons que nous traitons d'un AB Test sur un site de vidéo. La variante est la couleur de fond :

- 50% de la population a un fond bleu
- 50% de la population a un fond vert

On veut savoir quelle est la couleur que les utilisateurs préfèrent pour regarder les vidéos.

A chaque fois qu'un utilisateur regarde une vidéo, on enregistre les informations.

Nos données ressemblent à ça :

```
user1,blue,2
user2,green,5
user1,blue,3
user1,blue,1
user3,blue,1
```

Nous voulons savoir quel est le gagnant entre la couleur bleue et la verte.

6. UDF (User Defined Function)

Explications

Nous avons utilisé plusieurs fonctions jusqu'ici comme "max", "min", etc. Supposons que la fonction de "upper" n'existe pas. Avec Spark, nous pouvons l'ajouter comme suit :

```
val upper: String => String = word => word.toUpperCase

import org.apache.spark.sql.functions.udf
val upperUDF = udf(upper)

val diamonds = spark.read.option("header", "true").option("inferSchema",
"true").csv("diamonds.csv")
diamonds.withColumn("upperCut", upperUDF(col("cut"))).show
```

On pourrait aussi passer par une fonction :

```
def upper(word: String) = { word.toUpperCase }

import org.apache.spark.sql.functions.udf

val upperUDF = udf[String, String](upper)
```

Pour l'utiliser avec Spark SQL, il faudra l'enregistrer :

```
val upper: String => String = word => word.toUpperCase

spark.udf.register("myUpper", upper)

spark.sql("""select myUpper('iii')""").show
```

Une UDF peut aussi prendre plusieurs paramètres.

```
val printDescription: (String, String) => String = (name, age) => name + " is " +
age + " years old"

val people = ...

import org.apache.spark.sql.functions.udf
val printDescriptionUDF = udf(printDescription)

val result = people.withColumn("description", printDescription(col("name"),
col("age")))

result.show
```

Exercice 1

Nous allons créer une UDF dans un cas d'école : afficher la première lettre du "cut" des diamants suivie de la couleur.

Cela donnerait, pour "Ideal" comme "cut" et "E" comme couleur, "IE"

Exercice 2

Mais les UDFs, c'est coûteux.

Avec la liste des fonctions de sql, pouvez-vous imaginer une solution pour avoir le même résultat sans passer par une UDF ?

Liste des fonctions SQL

7. Les Window functions

Principes de base

Explications

Une fonction "window" effectue pour chaque ligne un calcul dans une "window".

Si on veut le prix moyen des diamants par couleur, mais aussi conserver le type de coupe, avec une window function, on peut. On pourrait obtenir quelque chose comme ça :

```
1500, "E", "Ideal" // 1500 correspond au prix moyen des diamants pour la couleur
"E"
1500, "E", "Premium" // 1500 correspond au prix moyen des diamants pour la couleur
"E"
2600, "J", "Ideal" // 2600 correspond au prix moyen des diamants pour la couleur
"J"
2600, "J", "Premium" // 2600 correspond au prix moyen des diamants pour la couleur
"J"
```

L'information que nous souhaitons calculer - le prix moyen des diamants par couleur - sera présente sur chaque ligne en fonction de la couleur.

C'est comme si nous avions les résultats d'un "group by" mais sur chaque ligne.

Avec Spark, les window functions s'utilisent soit via Spark SQL :

```
case class Salary(depName: String, empNo: Long, salary: Long)
val empsalary = Seq(
  Salary("sales", 1, 5000),
  Salary("personnel", 2, 3900),
  Salary("sales", 3, 4800),
  Salary("sales", 4, 4800),
  Salary("personnel", 5, 3500),
  Salary("develop", 7, 4200),
  Salary("develop", 8, 6000),
  Salary("develop", 9, 4500),
  Salary("develop", 10, 5200),
  Salary("develop", 11, 5200)).toDS

empsalary.createOrReplaceTempView("salary")

spark.sql("""
     select
        avg(salary) over (partition by depName) as avgSalary,
        depName,
        empNo,
        salary
     from
     salary """).show
```

```
+-----------------+---------+-----+------+
|        avgSalary|  depName|empNo|salary|
+-----------------+---------+-----+------+
|           5020.0|  develop|    7|  4200|
|           5020.0|  develop|    8|  6000|
|           5020.0|  develop|    9|  4500|
|           5020.0|  develop|   10|  5200|
|           5020.0|  develop|   11|  5200|
|4866.666666666667|    sales|    1|  5000|
|4866.666666666667|    sales|    3|  4800|
|4866.666666666667|    sales|    4|  4800|
|           3700.0|personnel|    2|  3900|
|           3700.0|personnel|    5|  3500|
+-----------------+---------+-----+------+
```

Soit via l'API DataFrame/Dataset :

```
val empsalary = ...
```

```
import org.apache.spark.sql.expressions.Window
val byDepName = Window.partitionBy("depName")

empsalary.withColumn("avgSalary", avg("salary") over byDepName).show
```

Exercice 1

Calculer le prix moyen des diamants par couleur avec une fonction "window" de manière à pouvoir afficher le reste des informations.

Exercice 2

Calculer le prix maximum et minimum des diamants par carat avec des fonctions "window" de manière à pouvoir afficher le reste des informations.

Utiliser les window functions pour faire du ranking

Les fonctions de "window" sont aussi utiles pour donner un rang aux éléments. On pourrait par exemple vouloir connaître les salaires des employés par département du plus grand au plus petit.

On peut y aller avec Spark SQL :

```
spark.sql("""
     select
        rank() over (partition by depName order by salary desc) as rankByDepName,
        depName,
        empNo,
        salary
     from
     salary """).show
```

```
+-------------+---------+-----+------+
|rankByDepName|  depName|empNo|salary|
+-------------+---------+-----+------+
|            1|  develop|    8|  6000|
|            2|  develop|   10|  5200|
|            2|  develop|   11|  5200|
|            4|  develop|    9|  4500|
|            5|  develop|    7|  4200|
|            1|    sales|    1|  5000|
|            2|    sales|    3|  4800|
|            2|    sales|    4|  4800|
|            1|personnel|    2|  3900|
|            2|personnel|    5|  3500|
+-------------+---------+-----+------+
```

Ou avec l'API DataFrame/Dataset :

```
val byDepnameSalaryDesc = Window.partitionBy('depname).orderBy('salary desc)
empsalary.withColumn("rankByDepName", rank() over byDepnameSalaryDesc).show
```

Exercice 1

On veut connaître le diamant qui coûte le plus cher par couleur. Que pouvons-nous remarquer dans les résultats ?

Exercice 2

Nous avons les données suivantes correspondant dans l'ordre au nom du produit, à sa catégorie et à l'argent que les ventes ont rapporté :

```
Thin, Cell phone, 6000
Normal, Tablet, 1500
Mini, Tablet, 5500
Ultra Thin, Cell phone, 6000
Very Thin, Cell phone, 5000
Big, Tablet, 2500,
Pro, Tablet, 4500
```

Avec une fonction de "window", à partir de celles-ci, on veut savoir quels sont les produits qui se vendent le mieux pour chaque catégorie.

Exercice 3

Il existe d'autres types de fonctions pour faire du window ranking comme "dense_rank()" ou "row_number()".

Vous pouvez chercher dans la documentation comment utiliser ces fonctions et générer vos propres cas de tests pour les explorer.

8. Manipulation de données encore plus avancée

Spark comprend encore de nombreuses fonctions dans les APIs haut niveau que nous n'avons pas encore vu comme "union", "collect_set", "collect_list", "intercept", "except".

Pour cette dernière partie, vous pouvez chercher dans la documentation comment utiliser ces fonctions et générer vos propres cas de tests pour les explorer.