Quelques notions de Spark, scala + ressources

1. Le terminal

Quelques commandes utiles pour le terminal (linux, mac OS)
> cd some/path/you/want/to/go : change directory
> cd .. : remonter d'un niveau dans l'arborescence
> ls : liste le contenu du répertoire courant
> pwd : savoir dans quel répertoire vous êtes
> mkdir : créer un répertoire
> cp : copy file
> mv : move file

2. Programmation Orientée Objet en scala

Si la programmation objet ne vous dit rien, je vous encourage vivement à en apprendre les concepts de bases ! Par exemple avec ce cours:
https://openclassrooms.com/courses/apprenez-a-programmer-en-python/premiere-approche-des-classes
C'est un cours en python, mais les notions de classes, attributs et méthodes sont similaires entre différents langages. Il existe de nombreux autres cours, videos, etc dans différents langages.

Un cours sur coursera pour en savoir plus sur scala:
https://openclassrooms.com/courses/apprenez-la-programmation-avec-scala/pourquoi-scala
https://www.coursera.org/learn/progfun1

Le point à retenir: Une méthode "method()" d'un objet (classe) "myObject" est appelée avec la syntaxe suivante:

```
> myObject.method(...éventuellement des variables ...)
```

3. Programmation fonctionnelle

Scala est aussi un langage de programmation fonctionnelle:
http://blog.xebia.fr/2015/05/22/decouvrir-la-programmation-fonctionnelle-1-fonctions/

C'est un concept de programmation puissant et offrant plus de robustesse que l'impératif ou la programmation objet. Scala n'est pas purement fonctionnel mais serait plutôt dans une zone grise entre les deux.

Le point à retenir concernant le fonctionnel avec scala Spark est la **composition de fonction**:

```
// Keeping urls that are associated with only one list of tags
val validUrls: DataFrame = dfInitial
 .select("cleanUrls", "tags")
 .dropDuplicates
 .groupBy("cleanUrls")
 .count
 .filter(col("count") === 1)
 .select("cleanUrls")
```

Dans l'exemple ci-dessus, `.select`, `.dropDuplicates`, `.groupBy-count` , `.filter` sont des fonctions qui s'appliquent au dataFrame précédant le point " `.` " et qui renvoient chacune un nouveau dataFrame. **On peut donc enchaîner ces fonctions**, autrement dit faire de la composition de fonction comme $(f \circ g \circ h)(x)$ ou noté autrement $f(g(h(x)))$. Dans l'exemple ci-dessus on fait $filter(dropDuplicates(select(dfInitial)))$.

4. Différences entre les RDD, les DataFrames et DataSets

Il est important de bien faire la distinction entre un RDD et un DataFrame en spark. Ce sont deux objets (classes) différents, qui permettent de faire des choses différentes.

Le RDD est dans les premières versions de spark la seule structure de données, il s'agit d'une liste de tuples, par exemple [(1, 2, 3, 4), (10, 20, 30, 40), …], distribuée. C'est à dire que les tuples sont répartis entre plusieurs exécuteurs (qui peuvent être sur différentes machines). Les RDD n'ont pas de notion de colonne, ils servent essentiellement à appliquer des traitements façon Map-Reduce.

Le DataFrame a été construit comme une surcouche sur le RDD. Il a une structure de colonne (voir la classe Column dans la doc) et se présente comme une table dans une base de donnée. Il offre de nouvelles fonctionnalités par rapport au RDD, et se rapproche des dataFrames de pandas en Python. Pour spark, en interne, le DataFrame est un RDD de Row (voir la classe Row dans la doc).

Le DataSet est une surcouche sur le DataFrame, son utilité est essentiellement d'apporter un typage plus fort des colonnes. Il est la structure par défaut à partir de Spark 2.0.0, toutes les méthodes des dataSets sont applicables aux DataFrames.

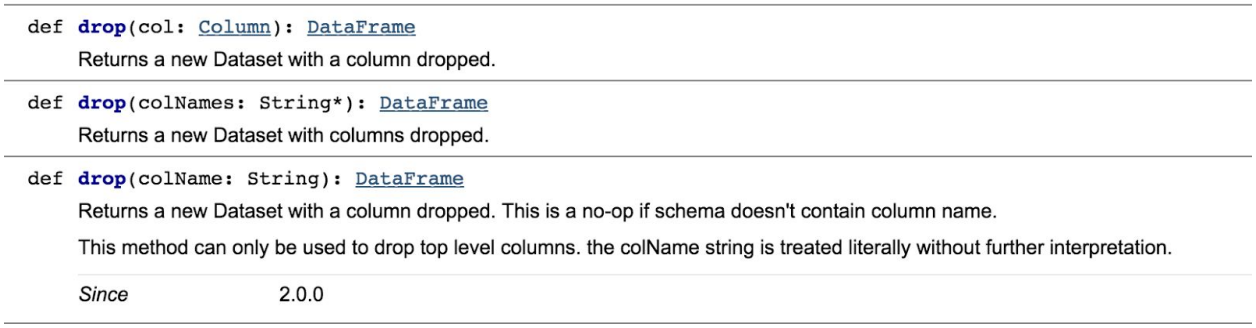
Le point à retenir: Utilisez les DataFrames le plus possible, et lorsque vous voulez appliquer des transformations sur vos dataFrames utilisez autant que possible les méthodes déjà implémentées dans Spark qui permettent de manipuler des objets Column (voir plus loin).

5. Savoir lire la doc

La doc de Spark contient un User Guide plus “user friendly”:
https://spark.apache.org/docs/latest/

Et une version détaillée de l’API Spark (dans différents langages), la version scala:
https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.package

Il est important d’apprendre à se servir de la doc détaillée, elle apporte des informations précieuses. Par exemple dans la page de la classe DataSet:



```
def drop(col: Column): DataFrame
    Returns a new Dataset with a column dropped.

def drop(colNames: String*): DataFrame
    Returns a new Dataset with columns dropped.

def drop(colName: String): DataFrame
    Returns a new Dataset with a column dropped. This is a no-op if schema doesn't contain column name.
    This method can only be used to drop top level columns. the colName string is treated literally without further interpretation.
    Since  2.0.0
```

Cet extrait de la doc nous donne plusieurs informations:

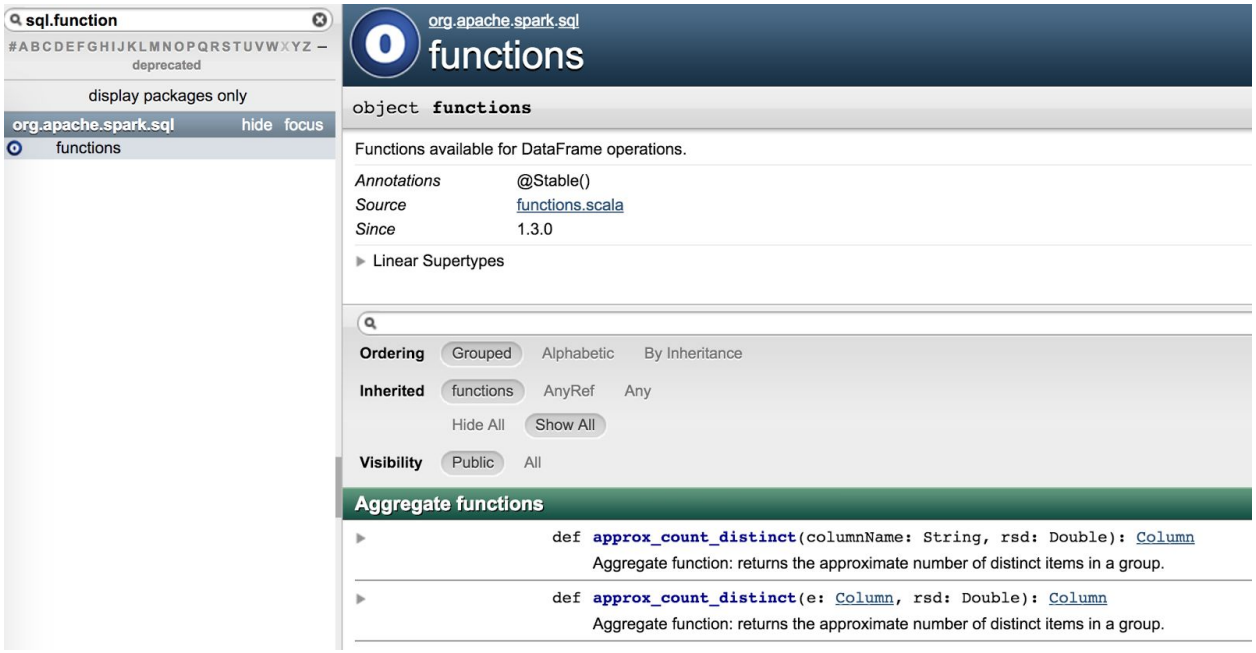
- Que les dataSets (et donc les dataFrames) ont une méthode drop(). Qu’on peut utiliser en faisant df.drop(...)

- Qu’il y a trois façon de se servir de cette méthode:
  - La 1ère attend un objet Column en Input. Ex:
    `df.drop($"country")` ou `df.drop(col("country"))`
  - La 2ème attend une série noms de colonnes (sous forme de String). Ex:
    `df.drop("country", "currency", "status")`
  - La 3ème attend un seul nom de colonne. Ex:
    `df.drop("country")`

- Que la méthode drop() retourne un DataFrame et donc qu’elle peut être enchaînée avec une autre méthode s’appliquant aux DataFrames (par exemple avec `.filter`).

Les noms de méthodes en bleus sont cliquables et ouvrent parfois des explications supplémentaires quand on clique dessus.

Si vous trouvez une fonction intéressante dans la documentation, le chemin vers la classe qui la contient est indiquée **en haut de la page**. Par exemple, sur la page de doc des fonctions *approx_count_distinct*, *lower*, *datediff, udf*, etc on voit:



Par conséquent, pour utiliser la fonction *approx_count_distinct,* il faudrait par exemple faire:

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

6. Importer des fonctions

La syntaxe pour importer une fonction en scala est:

- `import org.apache.spark.sql.functions.lower` si vous voulez importer uniquement la méthode “lower” depuis la classe sql.functions de spark
- `import org.apache.spark.sql.functions.{lower, datediff}` si vous voulez importer les méthodes lower et datediff de la classe sql.functions
- `import org.apache.spark.sql.functions._` si vous voulez importer toutes les méthodes de la classe sql.functions

7. Manipuler les colonnes d'un dataFrame

Une colonne d'un dataFrame spark est accessible en faisant $"col_name" ou col("name"). La notation avec $ est un raccourci de notation pratique, pour y avoir accès il faut ajouter `import spark.implicits._` à votre code APRES avoir définit une SparkSession que l'on aurait appelé ici "spark". Par exemple:

```
val spark = SparkSession
   .builder
   .getOrCreate()

import spark.implicits._
```

Pour savoir quelles opérations/transformations sont possibles sur des colonnes spark, lisez les pages :

- https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Column qui liste les méthodes appelables depuis un objet Column. Par exemple isNotNull:
    - `df.filter($"country".isNotNull)` ici isNotNull retourne "true" pour chaque ligne de la colonne "country" qui contient une valeur.

- https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$ qui liste les fonctions déjà implémentées dans spark et permettant d'appliquer des transformations sur les colonnes d'un dataFrame. Par exemple lower():
    - `df2.withColumn("country", lower($"country"))`
    - Ces fonctions font parties de la classe org.apache.spark.sql.functions

Si la transformation dont vous avez besoin n'est pas déjà implémentée dans Spark, vous pouvez coder vos propres fonctions sur des colonnes avec les UDF (User Defined Function): voir la page:
https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$

Pour créer vos UDFs il vous faut donc importer la fonction UDF de la classe sql.functions. La syntaxe pour définir une UDF est montrée dans l'exemple suivant:

```
import org.apache.spark.sql.functions.udf
// Define your udf:
def udf_value = udf{(name: String, value: Double) =>
 if (name == "plumbus" && value > 42.0)
   value
 else
   0.0
}
// To use the created UDF:
val df_modified: DataFrame = df.withColumn("value_modified",
udf_value($"object_name", $"price"))
```

Le type des colonnes “object_name” et “price” doit correspondre au type donné aux variables “name” et “value” dans la définition de l’UDF. L’UDF est appliquée à chaque ligne du dataFrame “df” en prenant successivement pour “name” et “value” les valeurs des colonnes “object_name” et “price” de chaque ligne. La fonction à l’intérieur de l’UDF peut être ce que vous voulez, et ici l’UDF prend en input deux colonnes, mais on peut en mettre autant que l’on veut.