TP6 Dataframes

Description des bases de données annuelles des accidents corporels de la circulation routière

Intro

Pour les exercices suivant on va utiliser une base de donnees publique du gouvernement qui contient des donnees sur les accidents de la route (https://www.data.gouv.fr/fr/datasets/base-de-donnees-accidents-corporels-de-la-circulation/ (https://www.data.gouv.fr/fr/datasets/base-de-donnees-accidents-corporels-de-la-circulation/))

Nous allons utiliser des dataframes et datasets pour manipuler ces donnees.

Environement

Vous pouvez faire ces excercices dans le spark shell.

Description des donnes

Pour chaque accident corporel (soit un accident survenu sur une voie ouverte à la circulation publique, impliquant au moins un véhicule et ayant fait au moins une victime ayant nécessité des soins), des saisies d'information décrivant l'accident sont effectuées par l'unité des forces de l'ordre (police, gendarmerie, etc.) qui est intervenue sur le lieu de l'accident.

Ces saisies sont rassemblées dans une fiche intitulée bulletin d'analyse des accidents corporels (BAAC). Cela comprend des informations de localisation de l'accident, telles que renseignées ainsi que des informations concernant les caractéristiques de l'accident et son lieu, les véhicules impliqués et leurs victimes.

Les bases de données de 2005 à 2015 sont désormais annuelles et composées de 4 fichiers (Caractéristiques – Lieux – Véhicules – Usagers) au format csv.

La description des differents fichiers se trouve ici: https://www.data.gouv.fr/s/resources/base-de-donnees-accidents-corporels-de-la-circulation/20160926-173908/Description_des_bases_de_donnees_ONISR_-Annees_2005_a_2015.pdf (https://www.data.gouv.fr/s/resources/base-de-donnees-accidents-corporels-de-la-circulation/20160926-173908/Description_des_bases_de_donnees_ONISR_-Annees_2005_a_2015.pdf)

Documentation

Documentation dataframes: http://spark.apache.org/docs/latest/sql-programming-guide.html#creating-dataframes (http://spark.apache.org/docs/latest/sql-programming-guide.html#creating-dataframes) Documentation API scala: http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset)

Dans les exercices suivants remplaceez les TODO par le code necessaire

Lecture et exploration des donnees CSV

```
import org.apache.spark.sql.SparkSession

def TODO = ??? // ceci est juste un marquer d'une valeur que vous devez remplacer dans les ligne

// on cree une session Spark avec un nom particulier pour la retrouver plus facilement dans le S
val mySession = SparkSession
  .builder()
  .config(new SparkConf()
             .setAppName(TODO))//rajouter comme parametre le nom de votre application
  .getOrCreate()
```

Took: 412 milliseconds, at 2016-12-12 20:55

```
// TODO: verifier/modifier le chemin vers les fichiers de donnees:
```

Took: 357 milliseconds, at 2016-12-12 20:49

```
// Lecture des donnes dans des DataFrames en utilisant la premier ligne du fichier comme nom des
val lieux /*:DataFrame*/ = mySession.read.option("header","true").format("com.databricks.spark.c
                                    .load(path + "/lieux_2015.csv")
                                    .cache
val caracteristiques = mySession.read.option("header","true").format("com.databricks.spark.csv")
                                     .load(path + "/caracteristiques_2015.csv")
                                     .cache
val usagers = mySession.read.option("header","true").format("com.databricks.spark.csv")
                                    .load(path + "/usagers_2015.csv")
                                    .cache
```

Took: 794 milliseconds, at 2016-12-12 20:49

```
// Un DataFrame est un DataSet[Row] ou par default chaque ligne est une "String"
```

Took: 477 milliseconds, at 2016-12-12 20:49

```
// afficher le schema des 3 autres Dataframes
```

```
// on peut utiliser la methode show pour afficher le contenu d'un DataFrame
```

Took: 1 second 722 milliseconds, at 2016-12-12 20:49

On peut suivre dans le SparkUI le plan d'execution de l'application de la methode show : http://localhost:4040 (http://localhost:4040) ainsi que la mise en cache des dataframes: http://localhost:4040/storage/ (http://localhost:4040/storage/) . Combien de lignes on a dans notre jeu de donnes ?

```
print {
  s"Le jeu de donnees contiens ${lieux.count} accidents qui ont implique ${vehicules.count} vehi
```

Took: 2 seconds 918 milliseconds, at 2016-12-12 20:49

```
//Afficher le nombre des victimes groupees par la gravite de la blessure. Utiliser un groupBy su
val victimes = TODO
```

Took: 1 second 631 milliseconds, at 2016-12-12 20:49

```
// Hmm, les valeurs numeriques ne sont pas trop comprehensible, nous pouvons les decoder en util
victimes.map{
  row => {
    val mapping = Map(
      "1" -> "Indemne",
      "2" -> "Tué",
      "3" -> "Blessé hospitalisé",
      "4" -> "Blessé léger"
    )
    mapping(row.getAs[String]("grav")) +" : "+ row.getAs[String]("count")
  }
```

Took: 2 seconds 361 milliseconds, at 2016-12-12 20:49

1. Quel est le nombre des blesses de la route en 2015?

27717

Took: 1 second 224 milliseconds, at 2016-12-12 20:49

2. Quelle est la repartition des victimes par age ?

Quel est le type de la colonne "an*_nais*" *du dataframe* "*_usagers*" ? Comment peut-on calculer la date de naissance?

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

// d'abord on doit creer une nouvelle colonne an_nais_int = pour convertir la colonne an_nais en
```

Took: 858 milliseconds, at 2016-12-12 20:49

```
// puis on defini une User Defined Function qui calcule l'age a partir de la date de naissance
val computeAge = udf {(an_nais: Int) =>  2016 - an_nais}

//et on rajoute une nouvelle colonne qui sera peuplee avec le resultat de l'application de l'udf
val usagersAge = usagers1.withColumn("age", computeAge(TODO) )
```

Took: 1 second 156 milliseconds, at 2016-12-12 20:49

```
// Afficher la repartition des victimes par age, triee par l'age des victimes
```

Took: 1 second 854 milliseconds, at 2016-12-12 20:49

Utiliser le SparkUI pour afficher le plan d'execution de la requete precedetnte http://localhost:4040 (http://localhost:4040)

4. Quelles sont les caracteristiques de 3 accidents le plus meurtriers de 2015 (date, nb vehicules, conditions meteo)

```
val accidents = usagers
    .filter($"grav" === 2 ) // on filtre les victimes
    .groupBy("Num_Acc") // on groupe sur l'id de l'accident
    .count() // on compte le nombre des victimes
    .sort(desc("count"))// on trie par le nombre des victimes
    .limit(3)// on garde uniquement les 3 accidents les plus meurtirers
    .withColumnRenamed("count","nb_victimes")
    .join(caracteristiques, usagers("Num_acc")=== caracteristiques("Num_Acc"))// on fait le join
    .join(lieux, usagers("Num_acc")=== lieux("Num_Acc"))// on fait le jointure avec les lieux
    .join(vehicules, usagers("Num_Acc")=== vehicules("Num_Acc")) // on fait la jointure avec les
    .groupBy(caracteristiques("an"),caracteristiques("mois"),caracteristiques("jour"),caracteris
    .count().withColumnRenamed("count","nb_vehicules") // on compte le nombre des vehicules
```

Took: 2 seconds 497 milliseconds, at 2016-12-12 20:49

Datasets (typer les DataFrames)

Dans les excercices precedents on a manipule des DataFrames = ensembles de lignes de chaines de caracteres. Les datasets nous permettent de rajouter de types au colonnes pour les manipuler plus facilement.

```
// On commence par definir une case class qui va definir le types des lignes de notre jeu de don
```

Took: 385 milliseconds, at 2016-12-12 20:49

```
//On doit transformer chaque ligne (Row) dans un objet Caracteristiques
// on defini une fonction de mapping qui prends un objet de type Row et retourne une Caracterist
def mapRow(r:Row): Caracteristiques = {
  Caracteristiques(
    r.getAs[String]("Num_Acc"),
    r.getAs[String]("lat").toDouble/100000,
    r.getAs[String]("long").toDouble/100000,
    r.getAs[String]("adr"),
    r.getAs[String]("col").toInt)
}
```

Took: 492 milliseconds, at 2016-12-12 20:49

```
// On transforme le dataframe caracteristique (Dataset[Row]) dans un DataSet caractDS (Dataset[C
val caractDS/*:Dataset[Caracteristiques]*/ = caracteristiques.filter(
            x => {
              try{
                mapRow(x)
                true
              }catch {
                case e: Exception => false
              }
```

Took: 943 milliseconds, at 2016-12-12 20:49

```
// On defnit une methode qui pour une Caracteristique retourne true si la Caracteristique decrit
def nearPlace(c:Caracteristiques): Boolean = {
   ( Math.abs(c.latitude - TODO)< 0.06 && Math.abs(c.longitude - TODO) < 0.06) //remplacez les T
}
nearPlace(Caracteristiques("test",48.71,2.24," my home", 0))// on test la methode en l'appliquan
```

true

Took: 589 milliseconds, at 2016-12-12 20:49

```
// Trouver un accident proche
```

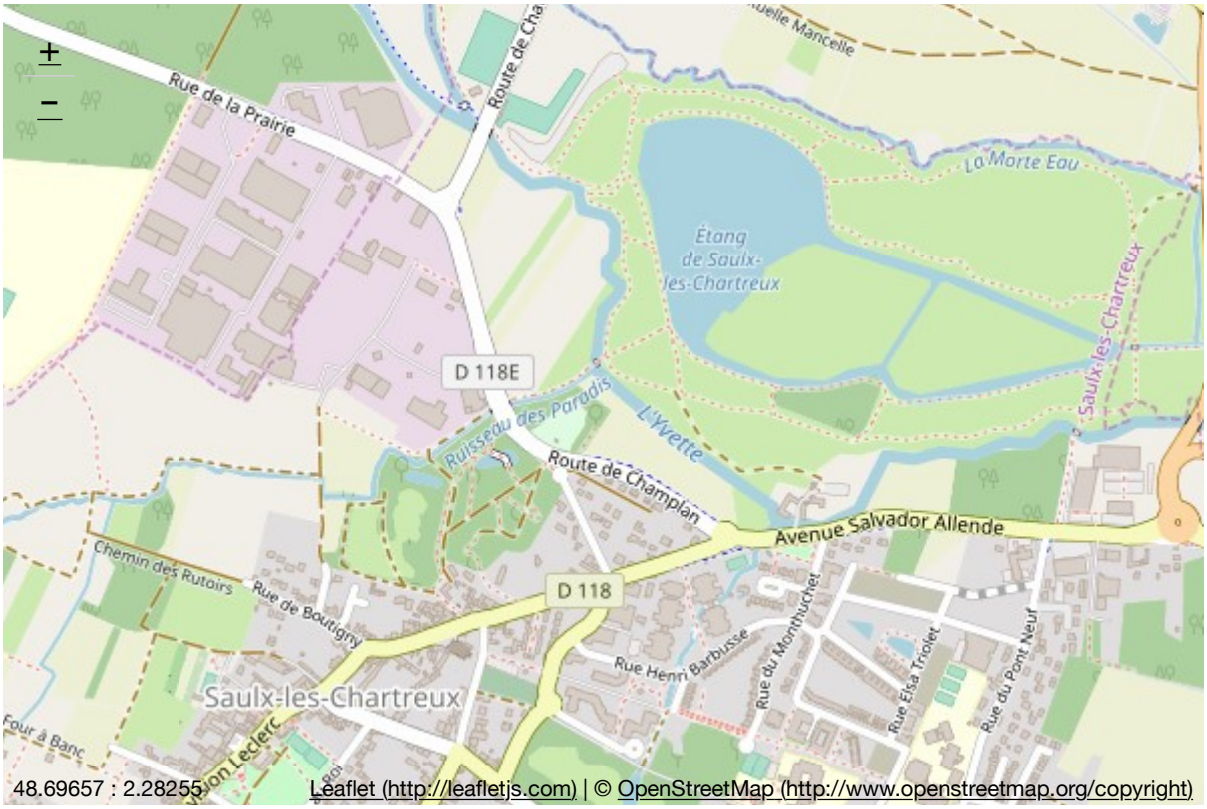
Took: 1 second 566 milliseconds, at 2016-12-12 20:49

5. [Optional] Exploration des donnes avec spark-notebook

```
// Trouver la liste des accidents proche de ma maison
```

Took: 1 second 364 milliseconds, at 2016-12-12 20:49

```
// Afficher les accidents proche sur une carte via un widget du notebook
val w = GeoPointsChart(points, maxPoints=500)
```



Took: 688 milliseconds, at 2016-12-12 20:49

```
// Via le widget PivotChart on peux faire des analyses comme dans un tableur
// TODO: faites des aggregations/heatmaps/charts en utilisant le widget PivotChart
```

300 or more entries total (Warning: showing only first 300 rows)
show/hide options

Heatmap | agg | lum | hrmn | gps | jour | long | col | int | an | adr

Count | dep

mois

mois \ dep	590	620	Totals
1	18	8	26
2	7	6	13
3	14	5	19
4	17	8	25
5	19	10	29
6	12	15	27
7	22	10	32
8	13	8	21
9	10	10	20
10	18	12	30
11	27	7	34
12	15	9	24
Totals	192	108	300