TP4 Big Data : Traiter des flux de données avec Spark

0. Mise en place

Pour ce dernier TP, vous allez utiliser Spark en local sur **votre VM ensai**. Vous trouverez sur Moodle une archive TP4 qui contient l'intégralité des fichiers utiles pour le TP. Téléchargez et décompressez la. Vous allez obtenir 3 dossiers

- server-python qui contient le code qui va créer un flux de données
- sparkqui contient les fichiers nécessaires à faire fonctionner spark en local
- data qui va contenir des données utilisées au cours du TP

Pour lancer spark :

```
%SPARK_HOME%/bin/pyspark --master local[4]  --driver-memory 500m
```

> ☕ Aide pyspark en console :
>
> - Pour coller utilisez maj+insert
> - Pour vider la console ctrl+l

1. Traiter des données depuis un flux TCP

Dans cette partie du TP vous allez traiter des données type IoT (*Internet of Things*). Imaginez qu'elle proviennent de montres connectées qui vont vous fournir diverses données sur des utilisateurs. Voici un exemple de données que vous pouvez récupérer :

```
{
"Arrival_Time": 1584786731698422519,
"Creation_Time": 1584786731485499800,
"Device": "nexus4_2",
"Index": 1039,
"Model": "nexus4",
"User": "b",
"gt": "null",
"x": 0.5642892523651901,
"y": -0.05573411422345695,
"z": 1.136185983195746
}
```

Bien sûr ces données sont générées aléatoirement et ne proviennent pas de vraies montres.

- 💻 Exécuter le fichier server.py.

  Ce script python va envoyer des données sur le port 9999 de votre ordinateur. Même si les données sont générées sur votre ordinateur, Spark va s'y connecter comme si elles étaient produite par un service distant.

- ✨ Exécuter pyspark

```
%SPARK_HOME%/bin/pyspark --master local[4]  --driver-memory 2g
```

- 🚧 Fixer le nombre de partition lors de la phase de shuffle à 5 pour spark SQL. Sans cela, Spark va générer 200 partitions pour vos données et ralentir fortement les traitements en local.

```
spark.conf.set("spark.sql.shuffle.partitions", 5)
```

- 📄 Importer les fonctions nécessaires pour la suite du TP

```
from time import sleep
from pyspark.sql.functions import from_json, window, col, expr
from pyspark.sql.types import StructType, StringType, LongType, DoubleType
```

- 📶 Ouvrir le flux TCP

```
tcp_stream = spark\
    .readStream\
    .format("socket")\
    .option("host", "127.0.0.1")\
    .option("port", "9999")\
    .load()
```

    - *format("socket")* : spécifie que le flux proviendra d'une socket TCP (pour faire très simple on récupéré des données produites par un serveur). Ce format de fichier est déconseillé pour des applications en production (comme une application utilisée par une entreprise pour traiter des données) car elle n'est pas tolérante à la faute. Préférez comme source des fichiers qui arrivent continuellement dans un dossier (dans ce cas la Spark ne se connecte pas à internet) ou une source Kafka (plus d'info)
    - *option("host", "127.0.0.1")* : la source des données se trouve sur le serveur appelé "127.0.0.1". Dans le cas présent cela signifie que c'est sur votre machine, mais pour ce connecter à une source distante il faut juste spécifier une autre adresse IP
    - *option("port", "9999")* : la source se trouve sur le port 9999.
    - *load()* : on ouvre le flux

> 🤔 Actuellement ce code ne va rien faire car n'oubliez pas spark est "lazy". Actuellement comme on n'utilise pas la source de données il ne s'y connecte pas

- 🖴 Afficher quelques données

```
raw_data_console = tcp_stream\
    .writeStream\
    .format('console')\
    .option('truncate', 'false')\
    .trigger(processingTime='3 seconds')\
    .start()

spark.streams.active #Voir la liste des streams en cours (c'est assez moche)

sleep(10) # on attend 10 seconde pour voir la console évoluer
raw_data_console.stop() # on ferme le stream. Si le stream continue après 10 secondes, c'est que vous n'avez pas validé la commande alors faite un "entrée"

```

  - *writeStream* : on va produire un stream
  - *format('console')* : que l'on écrit dans la console
  - *option('truncate', 'false')* : les colonnes sont affichée en entier. Vous pouvez tester sans pour voir la différence
  - *trigger(processingTime='3 seconds')* : on traite les données par paquet de 3 seconde. Vous pouvez faire varier ce nombre
  - *start()* : on dit (enfin) au stream de commencer

- 🖃 Afficher le schéma des données

```
tcp_stream.schema
```

Voilà la sortie que vous devez obtenir (en moins claire) :

```
StructType(List( # Liste de vos colones
    StructField(value,StringType,true) # Une seule colonne de type string
))
```

Actuellement les données sont traitées comme une seule est même colonnes "value" de type string. Pour bien les traiter , nous allons appliquer le bon schéma dessus

- ▶ Définir le schéma de nos données

```
schema = StructType()\
    .add('Arrival_Time',LongType(),True)\
    .add('Creation_Time',LongType(),True)\
    .add('Device',StringType(),True)\
    .add('Index',LongType(),True)\
    .add('Model',StringType(),True)\
    .add('User',StringType(),True)\
    .add('gt',StringType(),True)\
    .add('x',DoubleType(),True)\
    .add('y',DoubleType(),True)\
    .add('z',DoubleType(),True)
```

Comme les données proviennent d'un flux on ne peut pas inférer le schéma, donc on le définit à la main. Le True à la fin de chaque ligne spécifie que l'on accepte les valeurs null.

- 👇 Appliquer le schéma

```
df = tcp_stream.selectExpr('CAST(value AS STRING)')\
    .select(from_json('value', schema).alias('json'))\
    .select('json.*')
```

Pour appliquer le schéma on va appliquer une transformation à nos données.

  - *selectExpr('CAST(value AS STRING)')* : on cast la colonne value en string. C'est théoriquement déjà le cas, mais l'expliciter limite les erreurs.
  - *select(from_json('value', schema).alias('json'))* : on applique notre schéma à la colonne value, et on appelle cette nouvelle colonne 'json'
  - *select('json.*')* : on récupère uniquement les données de la colonne json
- ✅ Tester

```
json_data_console = df.writeStream\
    .format('console')\
    .trigger(processingTime='5 seconds')\
    .start()

sleep(10)
json_data_console.stop()
```

Ces commandes affichent dans la console le flux toutes les 5 secondes pendant 10 sec

> ✨ Il existe d'autres format d'output que la console pour les streams. Actuellement les données ne peuvent pas être utilisées par d'autres processus, ce qui est un problème pour la création d'une vraie application. A partir de maintenant nous allons enregistrer nos données en mémoire. Cela nous permettra d'utiliser nos données plus tard.

```
# La même chose mais ou on garde les données en mémoire pour les requêter plus tard.
json_data_memory = df.writeStream\
    .format("memory")\
    .queryName("stream")\
    .start()

# La partie requêtage. On va afficher les donner toutes les secondes. Remarquez que le stream dans la requête sql est le nom de la requête données plus haut.
for x in range(10):
    spark.sql("SELECT * FROM stream").show()
    sleep(1)


raw_data_memory.stop()

```

  - *format("memory")* : on stocke les données en mémoire
  - *queryName("stream")* : on donne un nom à la requête pour la réutiliser par la suite.
- ❌ Compter le nombre de ligne en erreur.

Notre source de données peut rencontrer des erreurs, et envoyer des données malgré une erreur dans le création. Nous allons les compter le nombre de ligne en erreur.

```
# On compte les input en erreur (tous les champs  sont null sauf arrival et creation time). On affiche directement le résultat en console
errorCount = df\
    .withColumn("error", expr("CASE WHEN Device IS NULL AND Index IS NULL AND Model IS NULL AND User IS NULL AND gt IS NULL THEN TRUE ELSE FALSE END"))\
    .groupBy("error")\
    .count()\
    .writeStream\
    .format("console")\
    .outputMode("complete")\
    .start()

sleep(10)
errorCount.stop()
```

    - *withColumn(...)* : on crée une colonnes error à partir d'une requête SQL
    - *groupBy("error")* : on fait un groupe by sur la colonne error
    - *count()* : on compte le nombre de ligne groupées
- 🔪 Filtrer les lignes en erreur. Une ligne est en erreur si une de ces valeurs vaut null

```
filterdDf = df\
    .na.drop("any")
```

    - na.drop("any") : on filtre toutes les lignes qui on une valeur null dans n'importe quelle colonne. Il est possible de filtrer les lignes qui ont uniquement des valeurs null avec all à la place d'any, ou de spécifier une liste à tester en faisant

```
.na.drop("all", subset=["col1", "col2"])
```

    - A partir de maintenant on va utiliser filterDf comme objet en entrée de nos streams.
- 🆗Tester que les erreurs sont bien filtrées

```
filterdDf_test = filterdDf \
    .withColumn("error", expr("CASE WHEN Device IS NULL AND Index IS NULL AND Model IS NULL AND User IS NULL AND gt IS NULL THEN TRUE ELSE FALSE END"))\
    .groupBy("error")\
    .count()\
    .writeStream\
    .format("console")\
    .outputMode("complete")\
    .start()

sleep(10)
filterdDf_test.stop()
```

Remarquez que la seule différence avec le comptage précédent est les données en entrée.

- 🚴 Compter le nombre de chaque activité

```
activityCounts = filterdDf.groupBy("gt").count()

activityCount_stream = activityCounts\
    .writeStream\
    .format("memory")\
    .outputMode("complete") \
    .queryName("activityCount_stream")\
    .trigger(processingTime='2 seconds')\
    .start()

# On attend 6 sec
for x in range(10):
    spark.sql("SELECT * FROM activityCount_stream").show()
    sleep(1)


activityCount_stream.stop()
```

    - *format("memory")* : on écrit le résultat en mémoire
    - *outputMode("complete")* : à chaque étape on met à jour intégralité des données en mémoire. Cela est utile quand on s'attend à ce que les données évolues au file du temps (comme lors d'un comptage). Il existe deux autres mode :
        - update : seule les lignes modifiées sont mises à jour. Mais la sortie doit supporter les opération de mise à jour de ligne (ce qui n'est pas le cas de la mémoire car spark stocke l'objet comme un dataset, et que les dataset ne peuvent pas être mise à jour, on peut seulement ajouter des lignes). La console par contre peut utiliser le mode update.
        - append : on ajoute les données au fur et à mesure à la sortie. Cela assure que les données sont traitées une seule fois (comportement par défaut)

```
# Le même conde avec une écriture dans la console et une format d'output update. Remplacer update par append voir la différence
activityCount_stream = activityCounts\
    .writeStream\
    .format("console")\
    .outputMode("update") \
    .trigger(processingTime='2 seconds')\
    .start()

activityCount_stream.stop()
```

> 🤔 Vous allez normalement voir apparaitre une ligne avec comme activité "null". Cela provient du fait que cette activité a pour nom la chaîne de caractère "null" et pas la valeur null

- ▼ Filtrer et sélectionner certaines infos

```
simpleTransformAndFilter = filterdDf.withColumn("stairs", expr("gt like '%stairs%'"))\
    .where("stairs")\
    .select("gt", "model", "arrival_time", "creation_time")\
    .writeStream\
    .queryName("simple_transform")\
    .format("memory")\
    .start()

for x in range(5):
    spark.sql("SELECT * FROM simple_transform").show()
    sleep(1)

simple_transform.stop()
```

  - *withColumn("stairs", expr("gt like '%stairs%'"))* : on crée un colonne stairs qui vaut TRUE si la colonne gt contient la chaîne de caractère "stairs" et FALSE sinon
  - *where("stairs")* : on garde que les lignes aves stairs == TRUE
  - *select("gt", "model", "arrival_time", "creation_time")* : on garde que les colonnes gt, model, arrival_time et creation_time

- 📊 Quelques stats

```
deviceMobileStats = filterdDf.cube("gt", "model")\
    .avg()\
    .drop("avg(Arrival_time)")\
    .drop("avg(Creation_time)")\
    .drop("avg(Index)")\
    .writeStream\
    .queryName("deviceMobileStats")\
    .format("memory")\
    .outputMode("complete")\
    .start()

for x in range(10):
    spark.sql("SELECT * FROM deviceMobileStats").show()
    sleep(1)

deviceMobileStats.stop()

```

La fonction cube prend une liste de colonnes en entrée (ici gt et model) et va faire tous les croisements possibles de ces variables et calculer les statistiques demandées (ici la moyenne) sur toutes les autres dimensions. Comme seules celles sur x, y et z nous intéresse on supprime les autres colonnes. Dans le tableau en sortie vous allez voir des valeurs "null" pour gt et model. Cela signifie que les moyennes ont était calculé sans prendre en compte cette dimension.

- ⏳ Utiliser les timestamps pour traiter les données en série temporelle
  - Conversion en timestamp

```
withEventTime = filterdDf.selectExpr(
    "*",
    "cast(cast(Creation_Time as double)/1000000000 as
timestamp) as event_time"
)

```

- On compte les évènements qui arrivent dans une fenêtre de 5 secondes

```
event_time =
withEventTime.groupBy(window(col("event_time"), "5
seconds")).count()\
    .writeStream\
    .queryName("event_per_window")\
    .format("memory")\
    .outputMode("complete")\
    .start()

for x in range(20):
    spark.sql("SELECT * FROM
event_per_window").show(50,False)
    sleep(1)

event_time.stop()

```

- On compte les évènements qui arrivent dans une fenêtre de 10 secondes avec des fenêtres toutes les 5 secondes

```
sliding_windows =
withEventTime.groupBy(window(col("event_time"), "10
seconds", "5 seconds")).count()\
    .writeStream\
    .queryName("event_per_window")\
    .format("memory")\
    .outputMode("complete")\
    .start()

for x in range(20):
    spark.sql("SELECT * FROM
event_per_window").show(50,False)
    sleep(1)

sliding_windows.stop()

```