

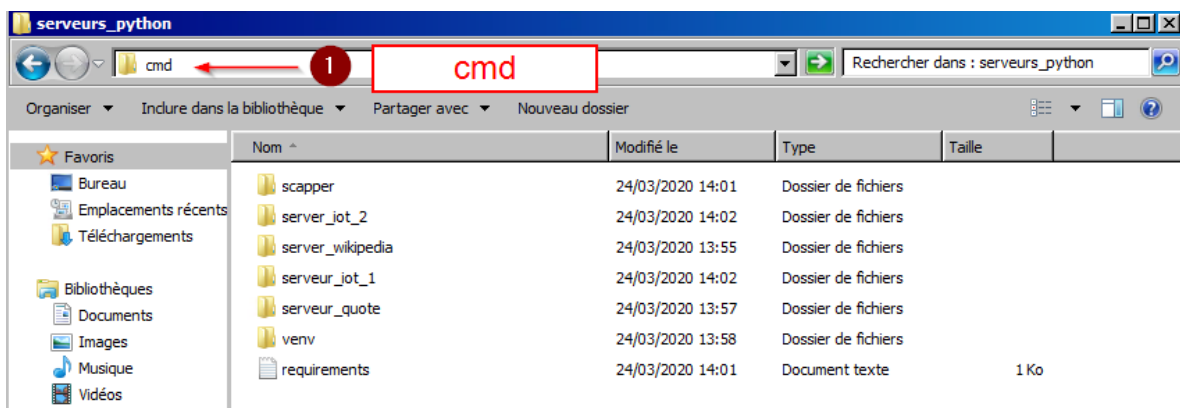
TP4 Big Data : Traiter des flux de données avec Spark

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 les fichiers suivants

- `serveurs_pythons`, qui contient les codes qui vont créer des flux de données
- `données_utilisateur`, qui contient des données statiques utiles dans le TP
- `hte-garonne`, qui contient des données statistiques utiles pour la partie notée
- `README.md`, qui est le sujet du TP au format markdown
- `raw_code.py`, qui contient les codes pour faire des copiés/collées

1. Allez dans le dossier `serveurs_pythons` et tapez `cmd` dans la barre d'adresse. Cela va vous ouvrir un invite de commande windows.



2. Installez les packages python nécessaire au fonctionnement des serveurs

```
1 | pip install -r requirements.txt --user --proxy http://pxcache-02.ensai.fr:3128
```

3. Lancez le serveur `server1`:

```
1 | python serveur_iot_1\server1.py
```

Gardez ce terminal ouvert ! Vous allez y voir apparaître les données au fur et à mesure qu'elles sont générées. Ce script python envoie des données sur le port `9999` de votre ordinateur.

4. Ouvrez un second terminal, depuis lequel vous lancez pyspark. Tout au long du TP vous garderez les terminaux ouverts, l'un qui génère les données, l'un qui réalise les traitements.

```
1 | %SPARK_HOME%/bin/pyspark --master local[4]
```

Explications:

- `%SPARK_HOME%/bin/pyspark` : on exécute le programme `pyspark`
- `--master local[4]` : on spécifie l'adresse du master ; `local` signifie que l'on va créer un cluster local ; `[4]` qu'on va demander 4 fils d'exécution distincts (qui pourront s'exécuter sur 4 processeurs distincts). Pour se connecter à un cluster existant, il suffirait de remplacer `local[4]` par l'adresse IP du master du cluster.
- 🛠 **Fixation du nombre de partitions pour la phase de *shuffle*** Une *partition* est, en Spark, le nom d'un bloc de données. Par défaut, Spark utilise 200 partitions (200 blocs) lors d'une phase de *shuffle*. Nos mini-batches de données seront tellement petits que cela n'a pas grand sens ici, et cela risque au contraire de ralentir les traitements.

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

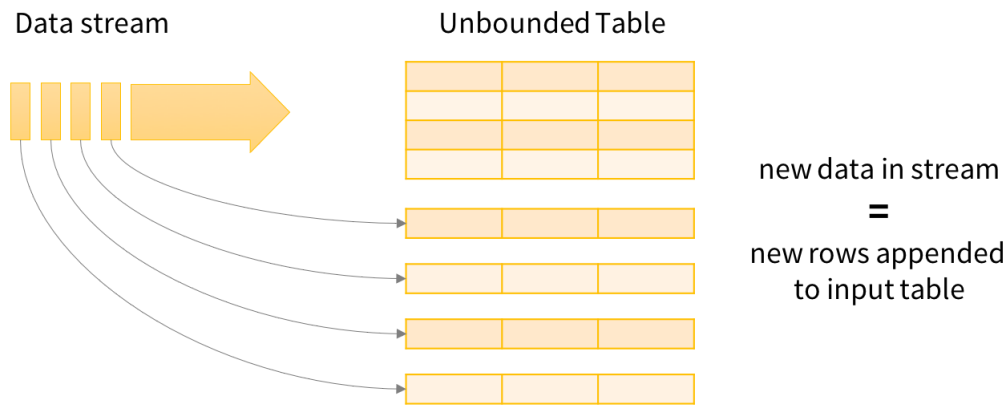
🤖 La phase de *shuffle* consiste à ré-ordonner les données selon leur clef, entre une étape *map* et une étape *reduce*. Par exemple, si vous comptez le nombre d'observations dans le groupe dans chaque groupe `g`, l'étape *map* consiste à compter le nombre de membres du groupe dans un bloc: `{g1:5, g2:10, g4:1, g5:3}` pour un bloc, `{g1:1, g2:2, g3:23, g5:12}` pour un autre. L'étape *shuffle* consiste à réorganiser les résultats intermédiaires en de nouveaux blocs, avec les mêmes clés au même endroit: `{g1:5, g1:1, g2:10, g2:2}` pour un bloc, `{g4:1, g5:3, g3:23, g5:12}` pour un autre. De cette façon, l'étape *reduce* est considérablement accélérée.

- 📄 Importer les fonctions nécessaires pour la suite du TP

```
1 | from time import sleep
2 | from pyspark.sql.functions import from_json, window, col, expr
3 | from pyspark.sql.types import StructType, StringType, LongType,
   | DoubleType, ShortType
```

1. Spark et les flux de données

Les flux de données sont arrivés progressivement dans Spark. En 2012, la fonctionnalité de traitement de flux de données (Spark Streaming, ou encore DStreams) est ajoutée à Spark, et rend possible le traitement de flux de données avec des fonctions hauts niveaux (comme *map* et *reduce*). En 2016 une nouvelle interface (ou API) est ajoutée, Structured Streaming, qui se base sur les DataFrames Spark et permet de manipuler des flux de données comme si des tableaux de données classiques.



Data stream as an unbounded Input Table

Traiter des données en flux oblige à les traiter au fil de l'eau. Quand une nouvelle observation est disponible, elle est traitée. Le traitement n'a donc pas réellement de fin, et la notion de "résultat" n'a pas grande sens: le résultat est mis à jour en permanence. Par exemple si vous comptez le nombre de tweets produits par heure avec un système de stream, tant qu'une heure n'est pas terminée vous allez avoir un nombre de tweets produit pour l'heure en cours qui augmente. Mais même à la fin de l'heure en question, certains tweets qui n'auront pas eu le temps de vous parvenir vont continuer à arriver, en retard. Peut-être que tel serveur a temporairement retenu une partie du trafic pendant qu'il redémarrait?

Les flux de données, ou *streams*, sont très utilisés. Voici quelques cas d'utilisation :

- **Alertes et notifications** : détection de fraude bancaire en temps réel ; suivi d'un réseau électrique grâce à des compteurs intelligents ; accompagnement médical d'une personne via des appareils de mesure connectés, etc.
- **Rapports en temps réel** : nombre d'utilisateur par minute d'un site ; portée d'une nouvelle campagne de publicité ; gestion automatique d'un portefeuille d'actions, etc.
- **ELT (extract transform load) incrémental** : des données non structurées arrivent en permanence et il faut les traiter (filtrer, mettre en forme) avant de les intégrer dans le système d'information de l'entreprise
- **Online machine learning** : des données sont transmises en permanence à un algorithme de machine learning qui améliore ses performances dynamiquement.

Traiter des données en flux présente malheureusement de nombreuses difficultés. En effet puisque le traitement n'a pas de fin, stocker les données indéfiniment génère nécessairement un problème de mémoire à terme. De même, traiter un évènement unique est simple, mais comment traiter une chaîne d'évènements ? (Ex: comment déclencher une alerte si on reçoit les valeurs 5, puis 6, puis 3 à la suite.) Dans un traitement classique, il suffirait de classer les données en fonction d'une variable temporelle mais, à cause la latence dans les transferts, il est possible de recevoir les données dans un ordre différent de l'émission!

Spark offre deux manière de traiter un flux de données: soit enregistrement par enregistrement (**one record at a time**), soit par ensemble d'enregistrements arrivés dans une fenêtre de temps (**micro batching**).

- Le **one record at a time** assure un faible délai entre l'arrivée d'un enregistrement et son traitement (**faible latence**), au prix d'une limite dans le débit maximal de données à traiter. Autrement dit, ce type de traitement est tout-ou-rien: tant que la quantité de données est faible, les données sont traitées en temps réel ; si le débit excède les

capacités de traitement, le système sera incapable de les gérer et le temps réel sera perdu.

- Le **micro batching**, quant à lui, traite les données toutes les `t` secondes. Cela veut dire que certaines données ne seront pas traitées immédiatement, mais avec le bénéfice de pouvoir affronter des débits beaucoup plus élevés. C'est le traitement généralement privilégié.

Pour avoir le meilleur arbitrage latence / débit, le mieux est de diminuer la taille des micro-batches jusqu'au moment où traiter un batch prend exactement le même temps que les données pour arriver. À partir de là, on remonte légèrement la taille des micro-batches pour atteindre à un point "optimal" entre débit et latence.

2. Traiter des données depuis un flux TCP

Dans cette partie du TP vous allez traiter des données type IoT (*Internet of Things*). Les données sont simulées par le serveur `server1`, et essaie d'imiter les données produites par une série de montres connectées. Les données, au format JSON, contiennent diverses données sur les propriétaires des montres :

```
1 {
2   "Arrival_Time": 1584786731698422519,
3   "Creation_Time": 1584786731485499800,
4   "Device": "nexus4_2",
5   "Index": 1039,
6   "Model": "nexus4",
7   "User": "b",
8   "gt": "null",
9   "x": 0.5642892523651901,
10  "y": -0.05573411422345695,
11  "z": 1.136185983195746
12 }
```

- 🖥 Vérifiez que votre serveur `server1` est toujours en activité. Si ce n'est pas le cas, retournez à la mise en place. Ce script python envoie 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.
- ✨ Vérifiez que votre autre terminal, qui exécute Spark, est toujours actif. Si ce n'est pas le cas, retournez à la mise en place.
- 📶 Ouvrir le flux TCP ([pour plus d'info sur les sources possibles](#))

```
1 tcp_stream = spark\
2   .readStream\
3   .format("socket")\
4   .option("host", "127.0.0.1")\
5   .option("port", "9999")\
6   .load()
```

- `format("socket")` : spécifie que le flux proviendra d'une socket TCP (pour faire très simple on récupère des données produites par un serveur). Ce format de fichier est déconseillé pour des applications *en production* ¹ car toute données perdue sera définitivement. Il n'est pas possible de en cas de redémarrage suite à une erreur de demander les messages des X dernières

minutes. Une façon plus sûre de transmettre un flux de donnée serait d'ajouter des fichiers continuellement dans un dossier (dans ce cas la Spark ne se connecte pas à internet) ou d'utiliser un service dédié comme Kafka ([plus d'info](#)).

- `option("host", "127.0.0.1")` : votre ordinateur, vu depuis votre ordinateur, possède l'adresse IP `127.0.0.1`. Pour se connecter à une source distante, il suffirait de remplacer cette adresse IP par celle de la source.
- `option("port", "9999")` : la source utilise le port 9999. Ce choix est purement conventionnel mais (1) il doit être le même pour l'émetteur et le récepteur et (2) il est déconseillé d'utiliser des ports "connus" pour éviter tout usage conflictuel du même port. Des exemples de ports connus sont les ports 20 et 21 utilisés pour les transferts de fichier FTP, 80 et 443 pour HTTP et HTTPS respectivement — voir [ici](#) pour une liste exhaustive.
- `load()` : on ouvre le flux

😬 Il ne se passe rien!? C'est normal! N'oubliez pas que Spark pratique l'évaluation paresseuse (*lazy evaluation*) : comme on n'utilise pas la source de données, il ne s'y connecte pas.

- 🖨 Afficher quelques données

```
1 raw_data_console = tcp_stream\  
2   .writeStream\  
3   .format('console')\  
4   .option('truncate', 'false')\  
5   .trigger(processingTime='3 seconds')\  
6   .start()  
7  
8 sleep(10)                # attendre 10 seconde pour voir la console  
                           évoluer  
9 raw_data_console.stop() # fermer le stream.  
10 # Si le stream continue après 10 secondes,  
11 # c'est que vous n'avez pas validé la commande!  
12 # Appuyez sur la touche "entrée".
```

- `writeStream` : on va produire un stream
 - `format('console')` : que l'on écrit dans la console
 - `option('truncate', 'false')` : les colonnes sont affichées 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

```
1 tcp_stream.schema
```

Voilà la sortie que vous devez obtenir :

```
1 StructType(List(StructField(value,StringType,true)))
```

Qu'est-ce que cela veut dire?

```

1 StructType(
2
3     List( # Liste de vos colonnes
4
5         StructField(value, StringType, true) # Une seule colonne de type
         string
6
7     )
8 )

```

Autrement dit, **les données ne sont pas correctement lues**. Chaque ligne est lue comme une seule grande chaîne de caractères. Pour bien les traiter, nous devons appliquer le bon **schéma**.

- ▶ Définir le schéma de nos données

```

1 schema_iot = StructType()\
2     .add('Arrival_Time', LongType(), True)\
3     .add('Creation_Time', LongType(), True)\
4     .add('Device', StringType(), True)\
5     .add('Index', LongType(), True)\
6     .add('Model', StringType(), True)\
7     .add('User', StringType(), True)\
8     .add('gt', StringType(), True)\
9     .add('x', DoubleType(), True)\
10    .add('y', DoubleType(), True)\
11    .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` (plus de détails [dans la documentation](#)).

- 🔗 Appliquer le schéma

```

1 iot_data = tcp_stream.selectExpr('CAST(value AS STRING)')\
2     .select(from_json('value', schema_iot).alias('json'))\
3     .select('json.*')

```

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

- `selectExpr('CAST(value AS STRING)')` : convertir la colonne `value` en chaîne de caractères. C'est théoriquement déjà le cas, mais l'explicitement 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

```

1  iot_data_console = iot_data\
2    .writeStream\
3    .format('console')\
4    .trigger(processingTime='5 seconds')\
5    .start()
6
7  sleep(10)
8  iot_data_console.stop() # copier aussi la ligne vide en-dessous!
9

```

Ces commandes affichent dans la console le flux toutes les 5 secondes pendant 10 sec. (Comme précédemment, si le flux continue, c'est que vous n'avez pas tapé **Entrée** après la dernière instruction. Par ailleurs, si les données sont toujours illisibles, élargissez la console Python!)

- ✨ Il existe d'autres format d'output que la console pour les flux de données, comme les formats fichiers (`csv`, `json`, etc.), le format mémoire ou le format Kafka. Dans la console, les données ne peuvent pas être utilisées par d'autres processus, ce qui est un problème pour la création d'une vraie application. À la place, nous allons enregistrer nos données en mémoire, ce qui nous permet d'utiliser nos données plus tard.

Remarque: le format mémoire est utilisé ici par souci de facilité. En pratique, il n'est utilisé que pour le débogage. En effet, une telle façon de procéder n'est possible que tant que les données n'excèdent pas les capacités physiques de l'ordinateur, puisque les données doivent tenir en mémoire! Mais alors pourquoi utiliser Spark dans ce cas?

```

1  # La même chose mais on garde les données en mémoire pour les
    requêter plus tard.
2  iot_data_memory = iot_data\
3    .writeStream\
4    .format("memory")\
5    .queryName("iot_data")\
6    .start()
7
8  # La partie requêtage. On va afficher les données toutes les
    secondes. Remarquez que le stream dans la requête sql est le nom de
    la requête données plus haut.
9  for x in range(10):
10     spark.sql("SELECT * FROM iot_data").show()
11     sleep(1)
12
13
14  iot_data_memory.stop()
15

```

- `format("memory")` : on stocke les données en mémoire dans un DataSet
- `queryName("iot_data")` : on donne un nom à la requête pour la réutiliser par la suite.
- ✖ Compter le nombre de ligne avec des erreurs.

Notre chaîne de traitement peut rencontrer des erreurs (caractères spéciaux ou autres problèmes de conversion...). Nous allons les compter le nombre de ligne avec une erreur, c'est-à-dire les variables qui ne sont pas renseignées à l'issue du traitement (`null` en SQL).

```


1  # On compte les input en erreur (tous les champs sont null sauf
   arrival et creation time). On affiche directement le résultat en
   console
2  errorCount = iot_data\
3      .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"))\
4      .groupBy("error")\
5      .count()\
6      .writeStream\
7      .format("console")\
8      .outputMode("complete")\
9      .start()
10
11 sleep(10)
12 errorCount.stop()

```

- `withColumn(...)` : on crée une colonne 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
- `outputMode("complete")` : à chaque nouveau mini-batch on met à jour l'intégralité des données en mémoire. Cela est utile quand on s'attend à ce que les données évoluent au fil du temps (comme lors d'un comptage). Il existe deux autres modes :
 - `"update"` : seules les lignes modifiées sont mises à jour. Cependant, la sortie choisie doit supporter les opérations 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 `dataSets` sont *immuables*). La console, par contre, peut utiliser le mode *update*.
 - `"append"` : on ajoute les résultats comme de nouvelles lignes ; les résultats précédents ne sont pas supprimés (comportement par défaut)

Les choses se compliquent puisque certaines sorties autorisent certaines méthodes ou non en fonction des opérations effectuées. Avec le cas des méthodes d'agrégation (compter, faire une moyenne par groupe, etc.) on a par exemple le tableau suivant ² :


Format	Agrégation	Mode
Console	Oui	update, complete
Console	Non	complete, append
Memory	Oui	complete
Memory	Non	append, complete

-  Filtrer les lignes avec erreur. Ici, on considérera qu'une ligne doit être écartée si une des variables n'est pas renseignée.

```

1  iot_filtered = iot_data\
2      .na.drop("any")

```



- `na.drop("any")` : on filtre toutes les lignes qui ont une valeur manquante (`null` en SQL) dans n'importe quelle colonne. Il aurait été possible de ne supprimer les observations avec *uniquement* des valeurs manquantes avec `na.drop("all")` ou de spécifier un ensemble de colonnes à considérer dans l'opération de filtrage avec `na.drop("<any or all>", subset=["col1", "col2"])`.
-  Tester que les erreurs sont bien filtrées

A partir de maintenant nous utiliserons `iot_filtered` en entrée de tous nos traitements.

```

1  errorCount = iot_filtered \
2      .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"))\
3      .groupBy("error")\
4      .count()\
5      .writeStream\
6      .format("console")\
7      .outputMode("complete")\
8      .start()
9
10 sleep(10)
11 errorCount.stop()
```

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

-  Compter le nombre de chaque activité

```

1  activityCounts = iot_filtered.groupBy("gt").count()
2
3  activityCount_stream = activityCounts\
4      .writeStream\
5      .format("memory")\
6      .outputMode("complete") \
7      .queryName("activityCount_stream")\
8      .trigger(processingTime='2 seconds')\
9      .start()
10
11 # Toutes les secondes pendant 10 secondes
12 # Remarquez qu'une fois sur deux rien ne se passe. En effet
13 # nous venons de spécifier une exécution toutes les 2 secondes
14 # (processingTime='2 seconds' ci dessus).
15 for x in range(10):
16     spark.sql("SELECT * FROM activityCount_stream").show()
17     sleep(1)
18
19 activityCount_stream.stop()
```

```

1  # Le même code avec une écriture dans la console et une format
   d'output update.
2  activityCount_stream = activityCounts\
3      .writeStream\
4      .format("console")\
5      .outputMode("update") \
6      .trigger(processingTime='2 seconds')\
7      .start()
8
9  activityCount_stream.stop()

```

- ▼ Filtrer et sélectionner certaines infos

```

1  only_stairs_activity = iot_filtered\
2      .withColumn("is_stair_activity", expr("gt like '%stairs%'"))\
3      .where("is_stair_activity")\
4      .select("gt", "model", "arrival_time", "creation_time")\
5      .writeStream\
6      .queryName("only_stairs_activity")\
7      .format("memory")\
8      .start()
9
10 for x in range(5):
11     spark.sql("SELECT * FROM only_stairs_activity").show()
12     sleep(1)
13
14 only_stairs_activity.stop()

```

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

- 📊 Quelques stats ([pour plus d'info](#))

```

1  deviceMobileStats = iot_filtered\
2      .select("x", "y", "z", "gt", "model")\
3      .cube("gt", "model")\
4      .avg()\
5      .writeStream\
6      .queryName("deviceMobileStats")\
7      .format("memory")\
8      .outputMode("complete")\
9      .start()
10
11 for x in range(10):
12     spark.sql("SELECT * FROM deviceMobileStats").show()
13     sleep(1)
14
15 deviceMobileStats.stop()
16

```

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. Dans le tableau en sortie vous allez voir des valeurs `null` pour `gt` et `model`. Cela signifie que les moyennes ont été calculées sans prendre en compte cette dimension.

- ⌚ Utiliser les timestamps pour traiter les données en série temporelle ([pour plus d'info](#))
 - Conversion en timestamp

```
1 | iot_filtered_withEventTime = iot_filtered.selectExpr(  
2 |     "*",  
3 |     "cast(cast(Creation_Time as double)/1000000000 as  
   | timestamp) as event_time"  
4 | )
```

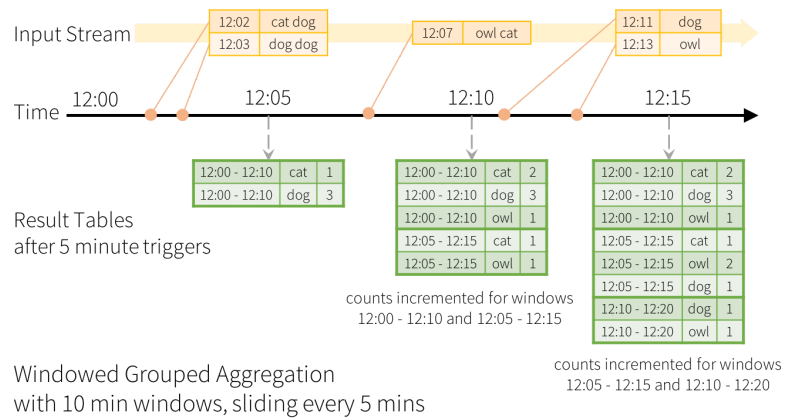
Mais que se passe-t-il ici?

```
1 | # DO NOT COPY ! THIS IS ONLY PEDAGOGICAL !  
2 |  
3 | # 1. creation_time est de type "double"  
4 | creation_time_as_double = cast(Creation_Time as double)  
5 |  
6 | # 2. diviser par 10^9 pour avoir le temps en seconde  
7 | # puis convertir au format "timestamp"  
8 | cast(creation_time_as_double/1000000000 as timestamp)
```

- On compte les événements qui arrivent dans une fenêtre de 5 secondes. Sauf que comme on reçoit un flux de données le temps qui nous importe n'est pas le temps d'arrivée mais le temps de création de la données (ici `event_time`). En effet les données n'arrivent pas forcément dans leur ordre de création à cause du temps de transmission.

```
1 | event_time = iot_filtered_withEventTime\  
2 |     .groupBy(window(col("event_time"), "5 seconds"))\  
3 |     .count()\  
4 |     .writeStream\  
5 |     .queryName("event_per_window")\  
6 |     .format("memory")\  
7 |     .outputMode("complete")\  
8 |     .start()  
9 |  
10 | for x in range(20):  
11 |     spark.sql("SELECT * FROM  
   | event_per_window").show(50, False)  
12 |     sleep(1)  
13 |  
14 | event_time.stop()  
15 |
```

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



```

1 event_sliding_windows = iot_filtered_withEventTime\
2   .groupBy(window(col("event_time"), "10 seconds", "5
3   seconds"))\
4   .count()\
5   .writeStream\
6   .queryName("event_per_window")\
7   .format("memory")\
8   .outputMode("complete")\
9   .start()
10
11 for x in range(20):
12     spark.sql("SELECT * FROM
13     event_per_window").show(50, False)
14     sleep(1)
15
16 event_sliding_windows.stop()

```

- Spark propose la possibilité de joindre un flux de données avec des données statiques

```

1 #Code pour charger un fichier csv
2 userData = spark.read.format("csv")\
3   .option("header", "true")\
4   .option("sep", ";")\
5   .option("inferSchema", "true")\
6   .load("chemin/de/mon/fichier/données utilisateurs.csv")
7
8 # On fait la jointure sur la colonne data, on ne souhaite pas avoir
9   les colonnes "Arrival_Time", "Creation_Time", "Index", "x", "y",
10   "z"
11 streamWithUserData = iot_filtered\
12   .drop("Arrival_Time", "Creation_Time", "Index", "x", "y", "z")\
13   .join(userData, ["User"])\
14   .writeStream\
15   .queryName("streamWithUserData")\
16   .format("memory")\
17   .start()
18
19 for x in range(10):
20     spark.sql("SELECT * FROM streamWithUserData").show(50, False)

```

```

20     sleep(1)
21
22     streamWithUserData.stop()

```

Une jointure avec une fonction agrégation

```

1  deviceMobileStatsUser = iot_filtered\
2    .join(userData, ["User"])\
3    .groupBy("User", "FirstName", "LastName")\
4    .count()\
5    .writeStream\
6    .queryName("stat_user")\
7    .format("memory")\
8    .outputMode("complete")\
9    .start()
10
11  # Le code suivant donne le même résultat (mais le plan d'exécution
12    est différent)
13  deviceMobileStatsUser = iot_filtered\
14    .groupBy("User")\
15    .count()\
16    .join(userData, ["User"])\
17    .writeStream\
18    .queryName("stat_user")\
19    .format("memory")\
20    .outputMode("complete")\
21    .start()
22
23  for x in range(5):
24      spark.sql("SELECT * FROM stat_user").show(50, False)
25      sleep(1)
26
27  deviceMobileStatsUser.stop()

```

3. À vous de jouer!

Votre but est de construire la première brique dans un tableau de bord pour Wikimedia, la maison mère de Wikipédia, afin de surveiller les articles de l'encyclopédie et de la défendre contre les pillages.

Wikimédia publie un flux de tous les changements qui ont lieu sur l'ensemble des plateformes à l'adresse suivante: <https://stream.wikimedia.org/v2/stream/recentchange>. Le format n'est pas adapté à Spark alors, comme précédemment, vous devez lancer un serveur qui lit, convertit et transfère le flux vers un port local auquel Spark peut s'abonner:

Chaque changement est un objet JSON de la forme suivante:

```

1  {
2    "$schema": "/mediawiki/recentchange/1.0.0",
3    "id": 1243377776,
4    "type": "categorize",
5    "namespace": 14,
6    "title": "Category:NA-importance India articles",
7    "comment": "[[:Category talk:1990s in Goa]] added to category",
8    "timestamp": 1585047749,

```

```


9      "user": "Jevansen",
10     "bot": false,
11     "server_url": "https://en.wikipedia.org",
12     "server_name": "en.wikipedia.org",
13     "server_script_path": "/w",
14     "wiki": "enwiki",
15     "parsedcomment": "<a href=\"/wiki/Category_talk:1990s_in_Goa\"
title=\"Category talk:1990s in Goa\">Category talk:1990s in Goa</a> added
to category"
16 }

```

Les variables qui nous intéressent sont: `title` (nom de la page), `user` (nom de l'utilisateur), `bot` (est-ce un robot qui a produit le changement), `timestamp` (à quel moment le changement a-t-il été produit), `wiki` (quel site de l'écosystème Wikimedia a été modifié).

1. Stockez ces informations (et uniquement celles-ci) dans un DataSet qui se mettra à jour toutes les 5 secondes
2. Combien de changements sont advenus depuis le début de notre abonnement, sur chaque site de Wikimedia? (vous afficherez le résultat dans la console)
3. Restreignez vous aux données de Wikipédia en français (`wiki=="frwiki"`). Maintenez à jour un DataSet qui donne le nombre d'édition (`type=="edit"`) dans une fenêtre glissante d'une heure calculée toutes les 5 minutes
4. Voici une liste de pages sensibles. Combien de modifications ont été effectuées sur l'une de ces pages depuis le début de l'abonnement?

4. Expert mode

 En plus des jointure stream x statique, il est également possible de faire des jointures entre streams ([pour plus d'info](#))

Pensez à lancer le server2 dans `serveurs_python/serveur_iot_2` (python `serveur_iot_2/server2.py`)

```

1  # Définition d'un nouveau stream
2  tcp_stream2 = spark\
3      .readStream\
4      .format("socket")\
5      .option("host", "127.0.0.1")\
6      .option("port", "10000")\
7      .load()
8
9  # Schema
10 schema_iot_2 = StructType()\
11     .add('Arrival_Time', LongType(), True)\
12     .add('Creation_Time', LongType(), True)\
13     .add('Device', StringType(), True)\
14     .add('Index', LongType(), True)\
15     .add('Model', StringType(), True)\
16     .add('User', StringType(), True)\
17     .add('bpm', ShortType(), True)\
18
19 # Mise au format plus filtrage
20 filtered_iot_2 = tcp_stream2.selectExpr('CAST(value AS STRING)')\
21     .select(from_json('value', schema_iot_2).alias('json'))\
22     .select('json.*')\

```

```

23     .na.drop("any")
24
25     # Jointure des deux flux selon la variable User
26     join_iot = iot_filtered.join(filtered_iot_2, "User").writeStream\
27         .format("console")\
28         .trigger(processingTime='10 seconds')\
29         .start()
30
31     join_iot.stop()

```

☹ Si vous faites bien attention joindre un flux avec un flux et joindre un flux avec des données statiques se font de la même façon car Spark manipule des DataFrame dans les deux cas.

🏠 Compter des mots de citations

🔗 On rentre dans les requêtes vraiment complexes donc ne passez pas beaucoup de temps sur cette partie

Lancer le fichier server_kaamelott.py. Ce serveur envoie des citations de kaamelott aux clients qui s'y connectent (il communique sur l'host 127.0.0.1 et le port 10001). Voici un exemple de donnée qu'il envoie :

```

1  {"character": "Karadoc", "quote": "Quand je pense à la chance que vous avez
    de faire partie d'un clan dirigé par des cerveaux du combat psychologique,
    qui se saignent... aux quatre parfums du matin au soir ! !"}

```

Et voici la requête pour obtenir le nombre de mot reçus par personnage

```

1  #Connexion au stream
2  kaamelott_stream = spark.readStream.format("socket").option("host",
    "127.0.0.1").option("port", "10001").load()
3
4  #Schéma
5  schema_kaamelott = StructType()\
6      .add('character', StringType(), True)\
7      .add('quote', StringType(), True)\
8
9  #Application du schéma
10 kaamelott_df = kaamelott_stream.selectExpr('CAST(value AS STRING)')\
11     .select(from_json('value', schema_kaamelott).alias('json'))\
12     .select('json.*')
13
14 # Le comptage de mots
15 # explode/split -> créent plusieurs lignes à partir d'une ligne (ici on
    coupe les mots avec split et on fait une ligne par mot de la citation)
16 # groupBy/count : on compte
17 words = kaamelott_df.select(kaamelott_df.character,
18     explode(
19         split(kaamelott_df.quote, ' ')
20     ).alias('word')
21 )\
22     .groupBy(kaamelott_df.character)\
23     .count()\
24     .writeStream\
25     .format("console")\
26     .outputMode("complete")\

```

```
27 | .trigger(processingTime='5 seconds')\  
28 | .start()
```

Pour plus d'information :

- [La doc spark officielle](#)
- ZAHARIA, B. C. M. (2018). *Spark: the Definitive Guide.* , O'Reilly Media, Inc. <https://proquest.safaribooksonline.com/9781491912201>
- <https://databricks.com/blog/2018/03/13/introducing-stream-stream-joins-in-apache-spark-2-3.html>
- <https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html>
- <https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html>

1. En production = en utilisation permanente, active. La production s'oppose au développement (la phase de construction d'un programme ou d'une application.) [↗](#)

2. Plus d'informations [dans la documentation spark.](#) [↗](#)