Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [153]:

```
NAME = "Vicent Ripoll Ramírez"
COLLABORATORS = ""
```

PEC 3: Noviembre 2022

Extracción de conocimiento de fuentes de datos heterogéneas mediante Spark SQL, RDDs y GraphFrames

En está práctica vamos a introducir los elementos que ofrece Spark para trabajar con estructuras de datos. Veremos desde estructuras muy simples hasta estructuras complejas, donde los campos pueden a su vez tener campos anidados. En concreto utilizaremos datos de twitter capturados en el contexto de las elecciones generales en España del 28 de Abril de 2019. La práctica está estructurada de la siguiente manera:

- **Parte 0:** Configuración del entorno
- **Parte 1:** Introducción a data frames estructurados y cómo operar extraer información *(2 puntos)*
  - **Parte 1.1:** Importar los datos *(0.25 puntos)*
  - **Parte 1.2:** *Queries* sobre sobre data frames complejos *(1.75 puntos)*
    - **Parte 1.2.1:** Queries SQL *(0.75 puntos)*
    - **Parte 1.2.2:** Queries sobre el pipeline *(1 punto)*
- **Parte 2:** Bases de datos HIVE y operaciones complejas *(3 puntos)*
  - **Parte 2.1:** Bases de datos Hive *(0.25 puntos)*
  - **Parte 2.2:** Más allá de las transformaciones SQL *(2.75 puntos)*
    - **Parte 2.2.1:** Tweets por población *(1.25 puntos)*
      - **Parte 2.2.1.1:** Utilizando SQL *(0.25 puntos)*
      - **Parte 2.2.1.2:** Utilizando RDD *(1 punto)*
    - **Parte 2.2.2:** Contar hashtags *(1.5 puntos)*
- **Parte 3:** Sampling *(2 Puntos)*
  - **Parte 3.1:** Homogéneo *(1 punto)*

    - **Parte 3.2:** Estratificado *(1 puntos)*
- **Parte 4**: Introducción a los datos relacionales *(2 puntos)*
    - **Parte 4.1:** Generar la red de retweets *(1.5 punto)*
        - **Parte 4.1.1**: Construcción de la edgelist *(0.75 puntos)*
        - **Parte 4.1.2**: Centralidad de grado *(0.75 puntos)*
    - **Parte 4.2:** Análisis de redes utilitzando GraphFrames *(0.5 punto)*
- **Parte 5**: Preguntas teóricas *(1 puntos)*

Parte 0: Configuración del entorno

In [3]:

```
import findspark
findspark.init()
```

In [4]:

```
import re
import os
import pandas as pd
from matplotlib import pyplot as plt
from math import floor
from pyspark import SparkConf, SparkContext, SQLContext, HiveContext
from pyspark.sql import Row
```

In [5]:

```
SUBMIT_ARGS = "--jars /opt/cloudera/parcels/CDH-6.2.0-1.cdh6.2.0.p0.967373/jars/graphframes

os.environ["PYSPARK_SUBMIT_ARGS"] = SUBMIT_ARGS

conf = SparkConf()
conf.setMaster("local[1]")
# Introducid el nombre de la app PEC3_ seguido de vuestro nombre de usuario
conf.setAppName("PEC3_vripollr")
sc = SparkContext(conf=conf)
```

Parte 1: Introducción a data frames estructurados i operaciones sobre ellos.

Como ya se ha mencionado, en esta práctica vamos ha utilizar datos de Twitter que recolectamos durante las elecciones generales en España del 28 de abril de 2019. Como veremos, los tweets tienen una estructura interna bastante compleja que hemos simplificado un poco en esta práctica.

Parte 1.1: Importar los datos

Lo primero que vamos ha aprender es cómo importar este tipo de datos a nuestro entorno. Uno de los tipos de archivos más comunes para guardar este formato de información es [la estructura JSON (https://en.wikipedia.org/wiki/JSON)](https://en.wikipedia.org/wiki/JSON). Esta estructura permite guardar información en un texto plano de diferentes objetos siguiendo una estructura de diccionario donde cada campo tiene asignado una clave y un valor. La estructura puede ser anidada, o sea que una clave puede tener como valor otra estructura de tipo diccionario.

Spark SQL permite leer datos de muchos formatos diferentes. Se os pide que leáis un fichero JSON (https://spark.apache.org/docs/2.4.0/sql-data-sources-json.html) de la ruta `/aula_M2.858/data/tweets28a_sample.json`. Este archivo contiene un pequeño *sample*, un 0.1% de la base de datos completa (en un siguiente apartado veremos cómo realizar este *sampleado*). En esta ocasión no se os pide especificar la estructura del dataframe ya que la función de lectura la inferirá automáticamente.

Esquema

```
sqlContext = SQLContext(sc)
tweets_sample = sqlContext.read.<FILL IN>

print("Loaded dataset contains %d tweets" % tweets_sample.count())
```

In [6]:

```
sqlContext = SQLContext(sc)
tweets_sample = sqlContext.read.json("/aula_M2.858/data/tweets28a_sample.json")

print("Loaded dataset contains %d tweets" % tweets_sample.count())
```

```
Loaded dataset contains 27268 tweets
```

In [7]:

```
assert tweets_sample.count() == 27268, "Incorrect answer"
```

El siguiente paso es mostrar la estructura del dataset que acabamos de cargar. Podéis obtener la información acerca de cómo está estructurado el DataTable utilizando el método `printSchema()` de la variable `tweets_sample`.

In [8]:

```
tweets_sample.printSchema()
```

```
root
 |-- _id: string (nullable = true)
 |-- created_at: long (nullable = true)
 |-- lang: string (nullable = true)
 |-- place: struct (nullable = true)
 |    |-- bounding_box: struct (nullable = true)
 |    |    |-- coordinates: array (nullable = true)
 |    |    |    |-- element: array (containsNull = true)
 |    |    |    |    |-- element: array (containsNull = true)
 |    |    |    |    |    |-- element: double (containsNull = true)
 |    |    |-- type: string (nullable = true)
 |    |-- country_code: string (nullable = true)
 |    |-- id: string (nullable = true)
 |    |-- name: string (nullable = true)
 |    |-- place_type: string (nullable = true)
 |-- retweeted_status: struct (nullable = true)
 |    |-- _id: string (nullable = true)
 |    |-- user: struct (nullable = true)
 |    |    |-- followers_count: long (nullable = true)
 |    |    |-- friends_count: long (nullable = true)
 |    |    |-- id_str: string (nullable = true)
 |    |    |-- lang: string (nullable = true)
 |    |    |-- screen_name: string (nullable = true)
 |    |    |-- statuses_count: long (nullable = true)
 |-- text: string (nullable = true)
 |-- user: struct (nullable = true)
 |    |-- followers_count: long (nullable = true)
 |    |-- friends_count: long (nullable = true)
 |    |-- id_str: string (nullable = true)
 |    |-- lang: string (nullable = true)
 |    |-- screen_name: string (nullable = true)
 |    |-- statuses_count: long (nullable = true)
```

Podéis observar que la estructura del tweet contiene múltiples campos anidados. Teneis que familiarizaros con esta estructura ya que será la que utilizaremos durante toda la práctica. Recordad también que no todos los tweets tienen todos los campos, como por ejemplo la ubicación (campo `place`). Cuando esto pasa el campo pasa a ser `NULL`. Podéis ver mas información sobre este tipo de datos en [este enlace (https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/tweet-object)](https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/tweet-object).

Parte 1.2: *Queries* sobre sobre data frames complejos

En está parte de la práctica veremos cómo hacer consultas sobre el dataset que acabamos de cargar. En la primera parte vamos ha utilizar [sentencias *SQL* (https://www.w3schools.com/sql/default.asp)](https://www.w3schools.com/sql/default.asp) (como las utilizadas en la mayoria de bases de datos relacionales). Para finalizar veremos un sistema alternativo para hacer consultas utilizando una aproximación más parecida a la que vistéis en la primera PEC.

Parte 1.2.1: Queries SQL

En primer lugar vamos a registrar nuestro DataFrame como una tabla de SQL llamado `tweets_sample`. Debido a que es posible que repitas esta práctica varias veces, vamos a tomar la precaución de eliminar cualquier tabla existente en primer lugar.

Podemos eliminar cualquier tabla SQL existente `tweets_sample` usando la sentencia SQL: `DROP TABLE IF EXISTS tweets_sample`. Para ejecutar un comando sql solo tenéis que utilizar el metodo SQL del objecto contexto, en este caso `sqlContext`.

Una vez ejecutado el paso anterior, podemos registrar nuestro DataFrame como una tabla de SQL usando [sqlContext.registerDataFrameAsTable()](https://spark.apache.org/docs/2.4.0/api/python/pyspark.sql.html#pyspark.sql.SQLContext.registerDataFrameAsT)

Esquema

```
sqlContext.sql(<FILL IN>)
sqlContext.registerDataFrameAsTable(<FILL IN>)
```

In [9]:

```
sqlContext.sql("DROP TABLE IF EXISTS tweets_sample")
sqlContext.registerDataFrameAsTable(tweets_sample, "tweets_sample")
```

In [10]:

```
assert sqlContext.sql("SELECT * FROM tweets_sample").count() == 27268, "Incorrecct answer"
```

Ahora se os pide que creeis una tabla `users_agg` con [la información agregada](https://www.w3schools.com/sql/sql_groupby.asp) de los usuarios que tengan definido su idioma (`user.lang`) como español (`es`). En concreto se os pide que la tabla contenga las siguientes columnas:

- **screen_name:** nombre del usuario
- **friends_count:** número máximo (ver nota) de personas a las que sigue
- **tweets:** número de tweets realizados
- **followers_count:** número máximo (ver nota) personas que siguen al usuario.

El orden en el cual se deben mostrar los registros es orden descendente acorde al número de tweets.

Nota: es importante que os fijéis que el nombre de *friends* i *followers* puede diferir a lo largo de la adquisición de datos. En este caso vamos ha utilizar la función de agregación `MAX` sobre cada uno de estos campos para evitar segmentar el usuario en diversas instancias.

Esquema

```
users_agg = sqlContext.sql(<FILL IN>)
users_agg.limit(10).show()
```

In [11]:

```
#users_agg = sqlContext.sql("""SELECT user.screen_name, COUNT(*) AS tweets FROM tweets_samp
#users_agg.limit(10).show()
users_agg = sqlContext.sql("""SELECT user.screen_name, MAX(user.friends_count) AS friends_c
                           COUNT(user.screen_name) AS tweets, MAX(user.followers_count) AS
                           FROM tweets_sample WHERE user.lang == 'es'
                           GROUP BY user.screen_name ORDER BY tweets DESC""")
users_agg.limit(10).show()
```

```
+---------------+-------------+------+---------------+
|    screen_name|friends_count|tweets|followers_count|
+---------------+-------------+------+---------------+
|       anaoromi|         6258|    16|           6774|
|    RosaMar6254|         6208|    14|           6245|
|        lyuva26|         3088|    13|           3732|
|PisandoFuerte10|         2795|    12|           1752|
|     carrasquem|          147|    12|            215|
|       jasalo54|         1889|    11|            689|
|  PabloChabolas|         4925|     9|           4042|
|      lolalailo|         4922|     9|           3738|
|     Lordcrow11|         5002|     9|           3069|
|    DuroBelinda|         5242|     9|           5778|
+---------------+-------------+------+---------------+
```

In [12]:

```
output = users_agg.first()
assert output.screen_name == 'anaoromi' and output.friends_count == 6258 and output.tweets
```

Imaginad ahora que queremos combinar la información que acabamos de generar con información acerca del número de veces que un usuario ha sido retuiteado. Para hacer este tipo de combinaciones necesitamos recurrir al `JOIN` de tablas (https://www.w3schools.com/sql/sql_join.asp). Primero debemos registrar la tabla que acabamos de generar en el contexto SQL. Recordad que primero debéis comprobar si la tabla existe y en caso afirmativo eliminarla. La tabla tenéis que registrarla bajo el nombre de `user_agg` .

In [13]:

```
sqlContext.sql("DROP TABLE IF EXISTS user_agg")
sqlContext.registerDataFrameAsTable(users_agg, "user_agg")
```

In [14]:

```
assert sqlContext.sql("SELECT * FROM user_agg").count() == 17925, "Incorrect answer"
```

Una vez registrada se pide que combinéis esta tabla y la tabla `tweets_sample` utilizando un `INNER JOIN` para obtener una nueva tabla con la siguiente información:

- ***screen_name:*** nombre de usuario
- ***friends_count:*** número máximo de personas a las que sigue
- ***followers_count:*** número máximo de personas que siguen al usuario.
- ***tweets:*** número de tweets realizados por el usuario.
- ***retweeted:*** número de retweets obtenidos por el usuario.

- ***ratio_tweet_retweeted:*** ratio de retweets por número de tweets publicados $\frac{retweets}{tweets}$

La tabla resultate tiene que estar ordenada de manera descendente según el valor de la columna `ratio_tweet_retweeted`

Esquema

```
retweeted = sqlContext.sql(<FILL IN>)

retweeted.limit(10).show()
```

In [15]:

```
retweeted = sqlContext.sql("""SELECT user_agg.screen_name, MAX(user_agg.friends_count) AS f
                           MAX(user_agg.followers_count) AS followers_count,
                           MAX(user_agg.tweets) AS tweets,
                           COUNT(tweets_sample.retweeted_status) AS retweeted,
                           COUNT(tweets_sample.retweeted_status)/MAX(user_agg.tweets) AS r
                           FROM tweets_sample
                           INNER JOIN user_agg
                           ON tweets_sample.retweeted_status.user.screen_name = user_agg.s
                           GROUP BY user_agg.screen_name ORDER BY ratio_tweet_retweeted DE
retweeted.limit(10).show()
#retweeted.limit(10).show()
#sqlContext.sql("SELECT user_agg.screen_name FROM user_agg").show(5)
#retweeted = sqlContext.sql("""SELECT user_agg.tweets, tweets_sample.retweeted_status.id
 #                          FROM tweets_sample
  #                          INNER JOIN user_agg
   #                         ON user_agg.screen_name = tweets_sample.screen_name""")
#sqlContext.sql("SELECT tweets_sample.retweeted_status FROM tweets_sample ORDER BY tweets_s
#sqlContext.sql("SELECT tweets_sample.retweeted_status FROM tweets_sample ORDER BY tweets_s
```

```
+--------------+-------------+---------------+------+---------+-------------
--------+
|   screen_name|friends_count|followers_count|tweets|retweeted|ratio_tweet_r
etweeted|
+--------------+-------------+---------------+------+---------+-------------
--------+
|          PSOE|        13635|         671073|     1|      155|
155.0|
|  CiudadanosCs|        92910|         511896|     1|      117|
117.0|
|     JuntsXCat|          202|          88515|     1|       73|
73.0|
|  PartidoPACMA|         1498|         232932|     1|       63|
63.0|
|  pablocasado_|         4567|         238926|     1|       50|
50.0|
|voxnoticias_es|         2146|          29582|     1|       44|
44.0|
|RaiLopezCalvet|         7579|          13574|     1|       43|
43.0|
|        iunida|        10225|         558318|     1|       39|
39.0|
|        Xuxipc|          311|         184967|     1|       37|
37.0|
|       Panik81|         1587|          15374|     1|       29|
29.0|
+--------------+-------------+---------------+------+---------+-------------
--------+
```

In [16]:

```
output = retweeted.first()
assert output.screen_name == 'PSOE' and output.friends_count == 13635 and output.tweets ==
```

Parte 1.2.2: Queries a través del pipeline

Las tablas de Spark SQL ofrecen otro mecanismo para aplicar las transformaciones y obtener resultados similares a los que se obtendría aplicando una consulta SQL. Por ejemplo utilizando el siguiente pipeline obtendremos el texto de todos los tweets en español:

```
tweets_sample.where("lang == 'es'").select("text")
```

Que es equivalente a la siguiente sentencia SQL:

```
SELECT text
FROM tweets_sample
WHERE lang == 'es'
```

Podéis consultar el API de spark SQL (https://spark.apache.org/docs/2.4.0/api/python/pyspark.sql.html) para encontrar más información sobre como utilitzar las diferentes transformaciones en tablas.

En este ejercicio se os pide que repliquéis la query obtenida en el apartado anterior empezando por generar la tabla `users_agg`. Podéis utilizar las transformaciones `where`, `select` (o `selectExpr`), `groupBy`, `count`, `agg` y `orderBy`

Esquema

```
users = tweets_sample.where(<FILL IN>).select(<FILL IN>)

users_agg = users.groupBy(<FILL IN>)\
                 .agg(<FILL IN>)\
                 .orderBy(<FILL IN>)

users_agg.limit(10).show()
```

In [17]:

```
users = tweets_sample.where("user.lang == 'es'").select("user.screen_name","user.friends_co

users_agg = users.groupBy("screen_name")\
                 .agg({"screen_name":"count","friends_count":"max","followers_count":"max"}
                 .orderBy("count(screen_name)",ascending=False)

users_agg.limit(10).show()
```

```
+---------------+-----------------+-----------------+-------------------+
|    screen_name|max(friends_count)|count(screen_name)|max(followers_count)|
+---------------+-----------------+-----------------+-------------------+
|       anaoromi|             6258|               16|               6774|
|    RosaMar6254|             6208|               14|               6245|
|        lyuva26|             3088|               13|               3732|
|PisandoFuerte10|             2795|               12|               1752|
|     carrasquem|              147|               12|                215|
|       jasalo54|             1889|               11|                689|
|  PabloChabolas|             4925|                9|               4042|
|      lolalailo|             4922|                9|               3738|
|     Lordcrow11|             5002|                9|               3069|
|    DuroBelinda|             5242|                9|               5778|
+---------------+-----------------+-----------------+-------------------+
```

In [18]:

```
output = users_agg.first()
assert output.screen_name == 'anaoromi'
```

Si os fijáis veréis que el nombre de las columnas no corresponde con el obtenido anteriormente, podéis cambiar el nombre de una columna determinada utilizando la transformación `withColumnRenamed`. Cambiad el nombre de las columnas para que coincidan con el apartado anterior y guardadlas en una variable `user_agg_new`.

Esquema

```
users_agg_new = users_agg.withColumnRenamed(<FILL IN>)\
                         .withColumnRenamed(<FILL IN>)\
                         .withColumnRenamed(<FILL IN>)

users_agg_new.limit(10).show()
```

In [19]:

```
users_agg_new = users_agg.withColumnRenamed("max(friends_count)","friends_count")\
                         .withColumnRenamed("count(screen_name)","tweets")\
                         .withColumnRenamed("max(followers_count)","followers_count")

users_agg_new.limit(10).show()
```

```
+---------------+-------------+------+---------------+
|    screen_name|friends_count|tweets|followers_count|
+---------------+-------------+------+---------------+
|       anaoromi|         6258|    16|           6774|
|    RosaMar6254|         6208|    14|           6245|
|        lyuva26|         3088|    13|           3732|
|PisandoFuerte10|         2795|    12|           1752|
|     carrasquem|          147|    12|            215|
|       jasalo54|         1889|    11|            689|
|  PabloChabolas|         4925|     9|           4042|
|      lolalailo|         4922|     9|           3738|
|     Lordcrow11|         5002|     9|           3069|
|    DuroBelinda|         5242|     9|           5778|
+---------------+-------------+------+---------------+
```

In [23]:

```
output = users_agg_new.first()
assert output.screen_name == 'anaoromi' and output.friends_count == 6258 and output.tweets
```

Cread ahora una tabla `user_retweets` utilizando transformaciones que contenga dos columnas:

- ***screen_name:*** nombre de usuario
- ***retweeted:*** número de retweets

Podéis utilizar las mismas transformaciones que en el ejercicio anterior. Ordenad la tabla en orden descendente utilizando el valor de la columna `retweeted`.

Esquema

```
    user_retweets = tweets_sample.<FILL IN>

    user_retweets.limit(10).show()
```

In [24]:

```
#user_retweets = tweets_sample.where("user.lang == 'es'").select("user.screen_name")

#users_agg = users.groupBy("screen_name")\
 #                .agg({"screen_name":"count"})\
  #               .orderBy("count(screen_name)",ascending=False).show()
#user_retweets = tweets_sample.where("user.lang == 'es'").select("retweeted_status.user.scr
 #               .groupBy("screen_name")\
  #               .agg({"screen_name":"count"})\
   #              .orderBy("count(screen_name)",ascending=False).show()
user_retweets = tweets_sample.select("retweeted_status.user.screen_name")\
                .groupBy("screen_name")\
                 .agg({"screen_name":"count"})\
                 .orderBy("count(screen_name)",ascending=False)\
                .withColumnRenamed("count(screen_name)","retweeted")

user_retweets.limit(10).show()
#sqlContext.sql("SELECT tweets_sample.retweeted_status FROM tweets_sample ORDER BY tweets_s
#quan sàpia com buscar els retweets es facil
#sqlContext.sql("SELECT * from tweets_sample").show(35,True)
#sqlContext.sql("SELECT tweets_sample.retweeted_status FROM tweets_sample ORDER BY tweets_s
```

```
+--------------+---------+
|   screen_name|retweeted|
+--------------+---------+
|        vox_es|      299|
|  ahorapodemos|      238|
| Santi_ABASCAL|      238|
|      iescolar|      166|
| AlbanoDante76|      161|
|          PSOE|      155|
|AntonioMaestre|      154|
|          KRLS|      149|
|         boye_g|      142|
|  CiudadanosCs|      117|
+--------------+---------+
```

In [25]:

```
output = user_retweets.first()
assert output.screen_name == 'vox_es' and output.retweeted == 299, "Incorrect output"
```

Otra manera de combinar dos tablas es utilizando el [metodo de tabla](https://spark.apache.org/docs/2.4.0/api/python/pyspark.sql.html) `join` (https://spark.apache.org/docs/2.4.0/api/python/pyspark.sql.html). Combinad la información de la tabla `users_agg_new` y `user_retweets` en una nueva tabla `retweeted` utilizando la columna `screen_name`. Ordenad la nueva tabla en orden descendente con el nombre de retweets.

Esquema

```
    retweeted = users_agg_new.join(<FILL IN>)\
                             .orderBy(<FILL IN>)

    retweeted.limit(10).show()
```

In [26]:

```
#users_agg_new.limit(10).show()
#user_retweets.limit(10).show()
#df1.join(df2, df1.screen_name==df2.screen_name)
retweeted=users_agg_new.join(user_retweets, users_agg_new.screen_name==user_retweets.screer
            .select(users_agg_new.screen_name, users_agg_new.friends_count, users_agg_new.f
            .orderBy("retweeted",ascending=False)
retweeted.limit(10).show()
```

```
+--------------+-------------+---------------+------+---------+
|   screen_name|friends_count|followers_count|tweets|retweeted|
+--------------+-------------+---------------+------+---------+
|          PSOE|        13635|         671073|     1|      155|
|  CiudadanosCs|        92910|         511896|     1|      117|
|     JuntsXCat|          202|          88515|     1|       73|
|  PartidoPACMA|         1498|         232932|     1|       63|
|  pablocasado_|         4567|         238926|     1|       50|
|voxnoticias_es|         2146|          29582|     1|       44|
|RaiLopezCalvet|         7579|          13574|     1|       43|
|        iunida|        10225|         558318|     1|       39|
|        Xuxipc|          311|         184967|     1|       37|
|       Panik81|         1587|          15374|     1|       29|
+--------------+-------------+---------------+------+---------+
```

In [27]:

```
output = retweeted.first()
assert output.screen_name == 'PSOE' and output.friends_count == 13635 and output.tweets ==
```

Notaréis que algunos de los registros que aparecen en la tabla `users_retweeted` no están presentes en la tabla retweeted. Esto es debido a que, por defecto, el método aplica un inner join y por tanto solo combina los registros presentes en ambas tablas. Podéis cambiar este comportamiento a través de los parámetros de la función.

Para terminar esta parte y reconstruir el resultado del apartado 1.2.1 vamos a añadir una columna `ratio_tweet_retweeted` con información del ratio entre retweets y tweets. Para ello debéis utilizar la transformación `withColumn`. El resultado debe estar ordenado considerando esta nueva columna en orden descendente.

Esquema

```
    retweeted = retweeted.withColumn(<FILL IN>)
    retweeted.limit(10).show()
```

In [31]:

```
#users_agg_new.limit(10).show()
#user_retweets.limit(10).show()
retweeted=retweeted.withColumn("ratio_tweet_retweeted",retweeted.retweeted/retweeted.tweets
```

In [32]:

```
output = retweeted.first()
assert output.screen_name == 'PSOE' and output.friends_count == 13635 and output.tweets ==
```

Parte 2: Bases de datos HIVE y operaciones complejas

Hasta ahora hemos estado trabajando con un pequeño sample de los tweets generados (el 0.1%). En esta parte de la PEC vamos a ver como trabajar y tratar con el dataset completo. Para ello vamos ha utilizar tanto transformaciones sobre tablas como operaciones sobre RDD cuando sea necesario.

Parte 2.1: Bases de datos Hive

Muchas veces los datos con los que vamos ha trabajar se van a utilizar en diversos proyectos. Una manera de organizar los datos es, en lugar de utilizar directamente los ficheros, recurrir a una base de datos para gestionar la información. En el entorno Hadoop una de las bases de datos más utilizadas es Apache Hive (https://hive.apache.org/), una base de datos que permite trabajar con contenido distribuido.

La manera de acceder a esta base de datos es creando un contexto Hive de manera muy similar a como declaramos un contexto SQL. Primero vamos a declarar un variable `hiveContext` instanciándola como un objeto de la classe `HiveContext` . Acto seguido vamos a comprobar cuantas tablas están registradas en este contexto.

Esquema

```
hiveContext = <FILL IN>
hiveContext.tables().show()
```

In [33]:

```
hiveContext = HiveContext(sc)
hiveContext.tables().show()
```

```
+--------+-----------------+-----------+
|database|        tableName|isTemporary|
+--------+-----------------+-----------+
| default|           d_pais|      false|
| default| d_tipo_habitacion|      false|
| default|     province_28a|      false|
| default|        tweets28a|      false|
| default|    tweets28afull|      false|
| default|tweets28a_sample25|      false|
| default|        user_info|      false|
| default|    user_info_old|      false|
|        |    tweets_sample|       true|
|        |         user_agg|       true|
+--------+-----------------+-----------+
```

Observad que ahora mismo tenemos siete tablas registradas en este contexto. Cinco de ellas no temporales y dos temporales, las que hemos registrado previamente. Por tanto sqlContext y hiveContext están concetados (es la misma sessión)

Vamos ha crear una variable `tweets` que utilizaremos para acceder a la tabla `tweets28a_sample25` guardada en `hiveContext` utilizando para ello el método `table()` de este objeto.

Esquema

```
tweets = <FILL IN>
print("Loaded dataset contains {} tweets".format(tweets.count()))
```

In [34]:

```
tweets = hiveContext.table("default.tweets28a_sample25")
print("Loaded dataset contains {} tweets".format(tweets.count()))
```

```
Loaded dataset contains 6354961 tweets
```

In [35]:

```
assert tweets.count() == 6354961, "Incorrect Answer"
```

Utilizando el mismo método que en el apartado 1.1, comprobad la estructura de la tabla que acabamos de cargar.

In [36]:

```
tweets.printSchema()
```

```
root
 |-- _id: string (nullable = true)
 |-- created_at: timestamp (nullable = true)
 |-- lang: string (nullable = true)
 |-- place: struct (nullable = true)
 |    |-- bounding_box: struct (nullable = true)
 |    |    |-- coordinates: array (nullable = true)
 |    |    |    |-- element: array (containsNull = true)
 |    |    |    |    |-- element: array (containsNull = true)
 |    |    |    |    |    |-- element: double (containsNull = true)
 |    |    |-- type: string (nullable = true)
 |    |-- country_code: string (nullable = true)
 |    |-- id: string (nullable = true)
 |    |-- name: string (nullable = true)
 |    |-- place_type: string (nullable = true)
 |-- retweeted_status: struct (nullable = true)
 |    |-- _id: string (nullable = true)
 |    |-- user: struct (nullable = true)
 |    |    |-- followers_count: long (nullable = true)
 |    |    |-- friends_count: long (nullable = true)
 |    |    |-- id_str: string (nullable = true)
 |    |    |-- lang: string (nullable = true)
 |    |    |-- screen_name: string (nullable = true)
 |    |    |-- statuses_count: long (nullable = true)
 |-- text: string (nullable = true)
 |-- user: struct (nullable = true)
 |    |-- followers_count: long (nullable = true)
 |    |-- friends_count: long (nullable = true)
 |    |-- id_str: string (nullable = true)
 |    |-- lang: string (nullable = true)
 |    |-- screen_name: string (nullable = true)
 |    |-- statuses_count: long (nullable = true)
```

Parte 2.2: Más allá de las transformaciones SQL

Algunas veces vamos a necesitar obtener resultados que precisan operaciones que van más allá de lo que podemos conseguir utilizando el lenguaje SQL. En esta parte de la práctica vamos practicar cómo pasar de una tabla a un RDD, para hacer operaciones complejas, y luego volver a pasar a una tabla.

Parte 2.2.1: Tweets por población

Parte 2.2.1.1: Utilizando SQL

Un pequeño porcentaje, alrededor del 1%, de los tweets realizados está geolocalizado. Eso quiere decir que para estos tweets tenemos información acerca del lugar donde han sido realizados guardado en el campo `place`. En este ejercicio se pide que utilizando una sentencia SQL mostréis en orden descendente cuántos tweets se han realizado en cada lugar. La tabla resultante `tweets_place` debe tener las siguientes columnas:

- ***name:*** nombre del lugar
- ***tweets:*** número de tweets

Recordad que no todos los tweets en la base de datos tienen que tener información geolocalizada, tenéis que filtrarlos teniendo en cuenta todos los que tienen un valor no nulo. La tabla tiene que estar en order descendente por numero de tweets.

Esquema

```
    tweets_place = hiveContext.sql(<FILL IN>)
    tweets_place.limit(10).show()
```

In [37]:

```
#users_agg = sqlContext.sql("""SELECT user.screen_name, MAX(user.friends_count) AS friends_
#                           COUNT(user.screen_name) AS tweets, MAX(user.followers_count) AS
 #                          FROM tweets_sample WHERE user.lang == 'es'
  #                         GROUP BY user.screen_name ORDER BY tweets DESC""")
#users_agg.limit(10).show()
#sqlContext.sql("SELECT * from tweets_sample").show(35,True)
#sqlContext.sql("SELECT tweets_sample.retweeted_status FROM tweets_sample ORDER BY tweets_s
#tweets_place = hiveContext.sql("""SELECT user.screen_name, MAX(user.friends_count) AS frie
 #                          COUNT(user.screen_name) AS tweets, MAX(user.followers_count) AS
  #                         FROM tweets_sample WHERE user.lang == 'es'
   #                        GROUP BY user.screen_name ORDER BY tweets DESC""")
#users_agg.limit(10).show()
#hiveContext.sql("SELECT place FROM tweets28a_sample25").show(35,True)

tweets_place = hiveContext.sql("""SELECT place.name AS name, COUNT(place.name) AS tweets
                           FROM tweets28a_sample25 WHERE place.name!='null'
                           GROUP BY place.name ORDER BY tweets DESC""")
tweets_place.limit(10).show()
hiveContext.sql("SELECT place.name FROM tweets28a_sample25 WHERE place.name!='null'")
```

```
+-----------+------+
|       name|tweets|
+-----------+------+
|     Madrid|  4911|
|  Barcelona|  3481|
|    Sevilla|   959|
|   Valencia|   689|
|   Zaragoza|   597|
|Villamartín|   570|
|     Málaga|   546|
|     Murcia|   461|
|      Palma|   416|
|   Alicante|   407|
+-----------+------+
```

Out[37]:

```
DataFrame[name: string]
```

In [38]:

```
output = tweets_place.first()
assert output.name == "Madrid" and output.tweets == 4911, "Incorrect output"
```

Parte 2.2.1.2: Utilizando RDD

Ahora se os pide que hagáis lo mismo pero esta vez utilizando RDD para realizar la agregación (recordad los ejercicios de contar palabras que hicisteis en la PEC 1).

El primer paso consiste en generar un tabla `tweets_geo` que solo contenga información de tweets geolocalizados con una sola columna:

- ***name:*** nombre del lugar desde donde se ha generado el tweet

Esquema

```
tweets_geo = <FILL IN>
```

In [39]:

```
tweets_geo=hiveContext.sql("SELECT place.name FROM tweets28a_sample25 WHERE place.name!='nu
tweets_geo.show()
tweets_geo.count()
```

```
+--------------------+
|                name|
+--------------------+
|           Barcelona|
|            Palencia|
|              Murcia|
|            Marbella|
|           Barcelona|
|             Camargo|
|            Calella|
|Sant Quirze del V...|
|San Cristóbal de ...|
|             Sevilla|
|           Wald (ZH)|
|           Wald (ZH)|
|              Madrid|
|              Seseña|
|                Vigo|
|Auditorio De La A...|
|              Madrid|
|              Madrid|
|              España|
|             Sevilla|
+--------------------+
only showing top 20 rows
```

Out[39]:

```
44477
```

In [40]:

```
assert tweets_geo.count() == 44477, "Incorrect answer"
```

Ahora viene la parte interesante. Una tabla puede convertirse en un RDD a través del atributo `.rdd` . Este atributo guarda la información de la tabla en una lista donde cada elemento es un objeto del tipo `Row` (https://spark.apache.org/docs/2.4.0/api/python/pyspark.sql.html#pyspark.sql.Row). Los objetos pertenecientes a esta clase pueden verse como diccionarios donde la información de las diferentes columnas queda reflejada en forma de atributo. Por ejemplo, imaginad que tenemos una tabla con dos columnas, nombre y apellido, si

utilizamos el atributo `.rdd` de dicha tabla obtendremos una lista con objetos del tipo row donde cada objeto tiene dos atributos: nombre y apellido. Para acceder a los atributos solo tenéis que utilizar la sintaxis *punto* de Python, e.g., `row.nombre` o `row.apellido` .

En esta parte del ejercicio se os pide que creeis un objeto `tweets_lang_rdd` que contenga una lista de tuplas con la información `(name, tweets)` sobre el nombre del lugar y el número de tweets generados desde allí. Recordad el ejercicio de contar palabras de la PEC 1.

Esquema

```
tweets_place_rdd = tweets_geo.<FILL IN>
```

In [41]:

```
tweets_place_rdd = tweets_geo.rdd.map(lambda x:(x,1)).reduceByKey(lambda a,b:a+b)
#tweets_place_rdd.count()
tweets_place_rdd.take(5)
```

Out[41]:

```
[(Row(name='Salas'), 3),
 (Row(name='Castilleja de la Cuesta'), 10),
 (Row(name='Toledo'), 124),
 (Row(name='Santa Eulária des Riu'), 9),
 (Row(name='Trentino-Alto Adige'), 2)]
```

In [42]:

```
assert tweets_place_rdd.count() == 4038, "Incorrect output"
```

Una vez generado este RDD vamos a crear un tabla. El primer paso es generar por cada tupla un objeto Row que contenga un atributo `name` y un atributo `tweets` . Ahora solo tenéis que aplicar el método `toDF()` para generar una tabla. Ordenad las filas de esta tabla por el número de tweets en orden descendente.

Esquema

```
tweets_place = tweets_place_rdd.<FILL IN>

tweets_place.limit(10).show()
```

In [43]:

```
tweets_place = tweets_place_rdd.map(lambda x:(x[0].name,x[1])).sortBy(lambda x:-x[1]).toDF(
tweets_place.limit(10).show()
#tweets_place = tweets_place_rdd
#tweets_place.collect()[1]
```

```
+-----------+------+
|       name|tweets|
+-----------+------+
|     Madrid|  4911|
|  Barcelona|  3481|
|    Sevilla|   959|
|   Valencia|   689|
|   Zaragoza|   597|
|Villamartín|   570|
|     Málaga|   546|
|     Murcia|   461|
|      Palma|   416|
|   Alicante|   407|
+-----------+------+
```

In [44]:

```
output = tweets_place.first()
assert output.name == "Madrid" and output.tweets == 4911, "Incorrect output"
```

Parte 2.2.2: Contar hashtags

En el ejercicio anterior hemos visto cómo podemos generar la misma información haciendo una agregación mediante SQL o utilizando RDDs. Como seguro que habéis observado la semántica de la sentencia SQL es mucho más limpia para realizar esta tarea. Pero no todas las tareas que os vais a encontrar se pueden hacer mediante sentencias SQL. En este ejercicio vamos a ver un ejemplo.

El objetivo de este ejercicio es contar el número de veces que cada hashtag (literalmente: palabras precedidas o serparadas por un #) ha aparecido en el dataset. Para evitar la sobrerrepresentación debida a los retweets vamos a concentrarnos en solo aquellos tweets que no son retweets de ningún otro, o dicho de otra manera, en aquellos en los que el campo `retweeted_status` es nulo. Cread una variable `non_retweets` que contenga todos estos tweets.

Esquema

```
non_retweets = <FILL IN>
```

In [45]:

```
#non_retweets=hiveContext.sql("""SELECT * FROM tweets28a_sample25 WHERE retweeted_status._i
 #                            retweeted_status.user.followers ='null' AND retweeted_status.
  #                           retweeted_status.user.id_str ='null' AND retweeted_status.use
   #                          retweeted_status.user.screen_name ='null' AND retweeted_statu
non_retweets=hiveContext.sql("SELECT text FROM tweets28a_sample25 WHERE retweeted_status IS
non_retweets.show()
non_retweets.count()
```

```
+--------------------+
|                text|
+--------------------+
|@Earco1977 Pues a...|
|@CastigadorY @jus...|
|Hola @policia, ho...|
|@LaRepublicaCat @...|
|@Podemos_CPRA @Ir...|
|@_GetMata_ @vox_e...|
|PSOE: RT SextaNoc...|
|@AnyPedrolo_2018 ...|
|PSOE: /❤Millones...|
|Hilo imprescindib...|
|Y TU, Pedro, ERE ...|
|@gabrielrufian Tr...|
|https://t.co/Ch8G...|
|@SextaNocheTV @PS...|
|@JoseNG1997 @popu...|
|@PSOE @mjmonteroc...|
|@Sericha10 @PSOE ...|
|@pujol_masip @Ine...|
|@Rafa_Hernando No...|
|Pablo Iglesias pr...|
+--------------------+
only showing top 20 rows
```

Out[45]:

```
1318664
```

In [46]:

```
assert non_retweets.count() == 1318664, "Incorrect answer"
```

Seguidamente vamos ha crear una variable `hashtags` que contenga una lista de tuplas con la información `(hashtag, count)`. Para ello, cread un RDD que contenga una lista con el texto de todos los tweets. Una vez hecho este paso tenéis que extraer los hashtags (palabras precedidas o separadas por un #) y contarlos.

Recordad los conocimientos adquiridos en la PEC 1 y el anterior ejercicio, os serán de gran ayuda.

In [48]:

```
#https://stackoverflow.com/questions/27826174/how-to-get-nth-row-of-spark-rdd
#non_retweets_rdd.collect()[1].text
#row=Row(age=11, name='Alice')
#row.age, row.name
#non_retweets_rdd=non_retweets.rdd.map(lambda x:(x,letter_counter(x.text,'#')))
def letter_counter(input,letter): #número de veces que aparece la letra 'letter' en la pala
    num_letters=0
    for char in input:
        if char==letter:
            num_letters=num_letters+1
    return num_letters
#non_retweets_rdd=non_retweets.rdd.map(lambda x:x.text).flatMap(lambda x:x.split(' ')).flat
#non_retweets_rdd=non_retweets.rdd.map(lambda x:x.text).flatMap(lambda x:x.split(' ')).filt
#non_retweets_rdd=non_retweets.rdd.map(lambda x:x.text).flatMap(lambda x:x.split(' ')).flat
non_retweets_rdd=non_retweets.rdd.map(lambda x:x.text).flatMap(lambda x:x.split(' ')).flatM
non_retweets_rdd.take(50)
```

Out[48]:

```
[('#F5', 1),
 ('#FASCISMONUNCAMAS', 4),
 ('#?”', 1),
 ('#SMARTPHONE?😉', 1),
 ('#VÉLEZBLANCO', 1),
 ('#TSHIRTS', 1),
 ('#EXTRALOUNGE', 3),
 ('#MESA', 1),
 ('#BADAJOZ', 31),
 ('#CS(LIBERALES…', 1),
 ('#EQUIPARACIONREAL', 4),
 ('#PUCHERAZO…', 1),
 ('#RECOMPTE', 1),
 ('#NACIONALISMOPROVINCIANOCATETO', 1),
 ('#MELODICESOMELOCUENTAS', 1),
 ('#HAYOTROCAMINO…', 1),
 ('#APRA:', 2),
 ('#NATIONALDEMOCRACY', 1),
 ('#LAAMERICAS', 1),
 ('#NAVIDAD', 2),
 ('#TEMASQUEIMPORTAN', 1),
 ('#PORTADAS', 36),
 ('#ABORTO', 23),
 ('#LIBERTADPARAASSANGE', 1),
 ('#VAMOSHUESCA', 3),
 ('#UNIDOSSOMOSMASFUERTES…', 3),
 ('#CISS', 1),
 ('#REFLEXIONESDELALMA…', 1),
 ('#NOCONMISIMPUESTOS…', 1),
 ('#LEYDHONT…', 1),
 ('#ALERTADORESDECORRUPCIÓN', 1),
 ('#AMNESIASELECTIVA', 1),
 ('#ABRERA', 3),
 ('#12INCH', 1),
 ('#MITINPPVIGO', 2),
 ('#LAESPAÑADELPLADUR', 13),
 ('#PCPE:', 2),
 ('#KATIEHOPKINS', 1),
```

```
('#TORREDEMBARRA…', 3),
('#CAMERA', 1),
('#DECORARTEHOGAR', 1),
('#10', 2),
('#VITORIA', 16),
('#FEMIFASCISMO', 1),
('#JUNTSXPALAU', 1),
('#DIÁLOGOSCONPATRICIA', 1),
('#CASADO;…', 1),
('#HAZCOMOROSA', 1),
('#COMMITTEES', 1),
('#BNG!', 1)]
```

Finalmente, se os pide que con el RDD obtenido generéis una tabla `hashtagsTable` compuesta de dos columnas:

- ***hashtag***
- ***num:*** número de veces que aparece cada hashtag.

Ordenadla en orden descendente por número de tweets.

Esquema

```
hashtagsTable = <FILL IN>

hashtagsTable.limit(20).show()
```

In [49]:

```
hashtagsTable=non_retweets_rdd.sortBy(lambda x:-x[1]).toDF().withColumnRenamed('_1','hashta
hashtagsTable.take(50)
```

Out[49]:

```
[Row(hashtag='#28A', num=39658),
 Row(hashtag='#ELDEBATEDECISIVO', num=28017),
 Row(hashtag='#ELDEBATEENRTVE', num=25640),
 Row(hashtag='#ELECCIONESGENERALES28A', num=8602),
 Row(hashtag='#EQUIPARACIONYA', num=7653),
 Row(hashtag='#ELECCIONESL6', num=7607),
 Row(hashtag='#HAZQUEPASE', num=6476),
 Row(hashtag='#DEBATEATRESMEDIA', num=5607),
 Row(hashtag='#VOX', num=5253),
 Row(hashtag='#DEBATERTVE', num=4622),
 Row(hashtag='#DEBATTV3', num=4501),
 Row(hashtag='#LAHISTORIALAESCRIBESTÚ', num=4417),
 Row(hashtag='#PORESPAÑA', num=3964),
 Row(hashtag='#VOTAPSOE', num=3855),
 Row(hashtag='#ELECCIONESGENERALES', num=3558),
 Row(hashtag='#28ABRIL', num=3474),
 Row(hashtag='#ILPJUSAPOL', num=3471),
 Row(hashtag='#ESPAÑAVIVA', num=3074),
 Row(hashtag='#PSOE', num=2757),
 Row(hashtag='#VALORSEGURO', num=2742),
 Row(hashtag='#ESPAÑA', num=2685),
 Row(hashtag='#LAESPAÑAQUEQUIERES', num=2258),
 Row(hashtag='#UNIDASPODEMOS', num=1967),
 Row(hashtag='#VOXSALEAGANAR', num=1807),
 Row(hashtag='#VAMOSCIUDADANOS', num=1708),
 Row(hashtag='#28A…', num=1577),
 Row(hashtag='#VOTAPORFAVOR', num=1484),
 Row(hashtag='#ELECCIONES2019', num=1291),
 Row(hashtag='#PP', num=1275),
 Row(hashtag='#VOXEXTREMANECESIDAD', num=1218),
 Row(hashtag='#STOPLEYABUSOSPOLICIALES', num=1174),
 Row(hashtag='#ESTAMOSMUYCERCA', num=1171),
 Row(hashtag='#SOYMUJERVOTOVOX', num=1057),
 Row(hashtag='#IDENTIDADVOX', num=1047),
 Row(hashtag='#CONRIVERANO', num=1029),
 Row(hashtag='#ELECCIONESGENERALES28ABRIL', num=1025),
 Row(hashtag='#SOSPRISIONES', num=1007),
 Row(hashtag='#YOVOTOUNIDASPODEMOS', num=997),
 Row(hashtag='#ELECCIONES28A', num=985),
 Row(hashtag='#DEBATEDECISIVOARV', num=984),
 Row(hashtag='#ELECCIONESGENERALES2019', num=976),
 Row(hashtag='#VOTA28A', num=900),
 Row(hashtag='#LAESPAÑAQUESUMA', num=870),
 Row(hashtag='#UNIDASPODEMOS28A', num=863),
 Row(hashtag='#VOTA', num=841),
 Row(hashtag='#YOVOTOVOX', num=840),
 Row(hashtag='#EQUIPARACIONYA…', num=812),
 Row(hashtag='#SOMOSUNOSOMOSVOX', num=779),
 Row(hashtag='#SISEPUEDE', num=768),
 Row(hashtag='#JORNADADEREFLEXION', num=765)]
```

In [50]:

```
output = hashtagsTable.first()
assert output.hashtag == "#28A" and output.num >= 38000, "Incorrect output"
```

Parte 3: Sampling

En muchas ocasiones, antes de lanzar costoso procesos, es una práctica habitual tratar con un pequeño conjunto de los datos para investigar algunas propiedades o simplemente para debugar nuestros algoritmos, a esta tarea se la llama sampling. En esta parte de la práctica vamos a ver los dos principales métodos de sampling y cómo utilizarlos.

Parte 3.1: Homogeneo

El primer sampling que vamos a ver es el homogeneo (https://en.wikipedia.org/wiki/Simple_random_sample). Este sampling se basta en simplemente escoger una fracción de la población seleccionando aleatoriamente elementos de la misma.

Primero de todo vamos ha realizar un sampling homogéneo del 1% de los tweets generados en periodo electoral sin reemplazo. Guardad en una variable `tweets_sample` este sampling utilizando el método `sample` descrito en la API de pyspark SQL (https://spark.apache.org/docs/2.4.0/api/python/pyspark.sql.html). El seed que vais a utilizar para inicializar el generador aleatorio es 42.

Esquema

```
seed = 42
fraction = 0.01

tweets_sample = tweets.<FILL IN>

print("Number of tweets sampled: {0}".format(tweets_sample.count()))
```

In [51]:

```
tweets_sample = tweets.sample(withReplacement=None,fraction=0.01,seed=42)
tweets_sample.take(1)
```

Out[51]:

```
[Row(_id='1117169844984610817', created_at=datetime.datetime(2019, 4, 13, 2
2, 57, 1), lang='es', place=None, retweeted_status=None, text='@Earco1977 Pu
es a mí me encantaría poder gobernar sin el PSOE.', user=Row(followers_count
=5893, friends_count=4783, id_str='509047038', lang='es', screen_name='Maria
_Beatle', statuses_count=15161))]
```

In [52]:

```
assert tweets_sample.count() == 63888, "Incorrect output"
```

Una de las cosas que resulta interesante comprobar acerca de los patrones de uso de las redes sociales es el patrón de uso diario. En este caso nos interesa el número promedio de tweets que se genera cada hora del día. Para extraer esta información lo que haremos primero, será generar una tabla `tweets_timestamp` con la información:

- ***created_at***: timestamp de cuando se publicó el tweet.
- ***hour***: a que hora del dia corresponde.
- ***day***: Fecha en formato MM-dd-YY

La función `hour` os servirá para extraer la hora del timestamp y la función `date_format` os permitirá generar la fecha. La tabla tiene que estar en orden ascendente según la columna `created_at`

In [53]:

```
#https://stackoverflow.com/questions/27826174/how-to-get-nth-row-of-spark-rdd
#non_retweets_rdd.collect()[1].text
#row=Row(age=11, name='Alice')
#row.age, row.name
#non_retweets_rdd=non_retweets.rdd.map(lambda x:(x,letter_counter(x.text,'#')))

#hashtagsTable=non_retweets_rdd.sortBy(lambda x:-x[1]).toDF().withColumnRenamed('_1','hasht
#hashtagsTable.take(50)

#date_format(tweets_sample.collect()[1].created_at,"yyyy MM dd") #no funciona
#tweets_sample.rdd.map(lambda item: (item[1].date(),item[1].hour,item[1].from_utc_timestamp
#map(lambda x:(x[0].datatime.datatime,x[1]))
#tweets_sample.collect()[1].lang

#retweeted = sqlContext.sql("""SELECT user_agg.screen_name, MAX(user_agg.friends_count) AS
 #                           MAX(user_agg.followers_count) AS followers_count,
  #                          MAX(user_agg.tweets) AS tweets,
   #                         COUNT(tweets_sample.retweeted_status) AS retweeted,
    #                        COUNT(tweets_sample.retweeted_status)/MAX(user_agg.tweets) AS
     #                       FROM tweets_sample
      #                      INNER JOIN user_agg
       #                     ON tweets_sample.retweeted_status.user.screen_name = user_agg.
        #                    GROUP BY user_agg.screen_name ORDER BY ratio_tweet_retweeted D

#output=sqlContext.sql("SELECT created_at, hour(created_at) AS hour, date(created_at), crea
#output.hour

from pyspark.sql.functions import date_format, hour, from_utc_timestamp

sqlContext.registerDataFrameAsTable(tweets_sample, "tbl")
tweets_timestamp=sqlContext.sql("""SELECT created_at, hour(created_at) AS hour,
                                date_format(date(created_at), "0M-d-YY") AS day FROM tbl
                                ORDER BY created_at""")

tweets_timestamp.limit(20).show()
```

```
+-------------------+----+--------+
|         created_at|hour|     day|
+-------------------+----+--------+
|2019-04-12 03:06:25|   3|04-12-19|
|2019-04-12 05:08:47|   5|04-12-19|
|2019-04-12 09:15:02|   9|04-12-19|
|2019-04-12 09:38:31|   9|04-12-19|
|2019-04-12 10:46:02|  10|04-12-19|
|2019-04-12 10:47:17|  10|04-12-19|
|2019-04-12 10:49:17|  10|04-12-19|
|2019-04-12 10:49:30|  10|04-12-19|
|2019-04-12 10:54:29|  10|04-12-19|
|2019-04-12 10:55:06|  10|04-12-19|
|2019-04-12 10:55:08|  10|04-12-19|
|2019-04-12 10:55:16|  10|04-12-19|
|2019-04-12 10:55:28|  10|04-12-19|
|2019-04-12 10:55:52|  10|04-12-19|
|2019-04-12 10:58:11|  10|04-12-19|
|2019-04-12 10:59:53|  10|04-12-19|
|2019-04-12 11:00:27|  11|04-12-19|
|2019-04-12 11:00:34|  11|04-12-19|
|2019-04-12 11:00:54|  11|04-12-19|
|2019-04-12 11:02:18|  11|04-12-19|
```

```
+-------------------+----+--------+
```

In [54]:

```
output = tweets_timestamp.first()
assert output.day == "04-12-19" and output.hour == 3, "Incorrect output"
```

El paso siguiente es agregar estos datos por hora y día en una tabla `tweets_hour_day` . Tenéis que crear una tabla `tweets_hour` con la información:

- ***hour:*** hora del dia
- ***day:*** fecha
- ***count:*** número de tweets generados

Ordenad la tabla por orden descendente de tweets.

In [55]:

```
#retweeted = sqlContext.sql("""SELECT user_agg.screen_name, MAX(user_agg.friends_count) AS
 #                            MAX(user_agg.followers_count) AS followers_count,
  #                           MAX(user_agg.tweets) AS tweets,
   #                          COUNT(tweets_sample.retweeted_status) AS retweeted,
    #                         COUNT(tweets_sample.retweeted_status)/MAX(user_agg.tweets) AS
     #                        FROM tweets_sample
      #                       INNER JOIN user_agg
       #                      ON tweets_sample.retweeted_status.user.screen_name = user_agg.
        #                     GROUP BY user_agg.screen_name ORDER BY ratio_tweet_retweeted D

tweets_hour_day=sqlContext.sql("""SELECT hour(created_at) AS hour,
                date_format(date(created_at), "0M-d-YY") AS day,
                COUNT(_id ) as count  FROM tbl GROUP BY day, hour
                ORDER BY count DESC""")
tweets_hour_day.limit(20).show()
```

```
+----+--------+-----+
|hour|     day|count|
+----+--------+-----+
|  23|04-23-19| 1222|
|  23|04-22-19| 1116|
|   0|04-24-19|  999|
|  23|04-28-19|  929|
|  22|04-23-19|  917|
|  22|04-22-19|  913|
|  22|04-28-19|  856|
|   0|04-29-19|  838|
|   0|04-23-19|  713|
|  21|04-28-19|  494|
|   1|04-24-19|  427|
|  16|04-28-19|  417|
|  15|04-28-19|  417|
|  20|04-28-19|  412|
|   1|04-29-19|  408|
|  18|04-28-19|  393|
|  17|04-28-19|  386|
|  19|04-28-19|  366|
|  21|04-23-19|  354|
|  14|04-28-19|  346|
+----+--------+-----+
```

In [56]:

```
output = tweets_hour_day.first()
assert output.hour == 23 and output['count'] == 1222, "Incorrect output"
```

Por último solo nos queda hacer una agregación por hora para conseguir el promedio de tweets por hora. Tenéis que generar una tabla `tweets_hour` con la información:

- ***hour:*** Hora
- ***tweets:*** Promedio de tweets realizados

Recordad que estamos trabajando con un sample del 1% por tanto tenéis que corregir la columna `tweets` para que refleje el promedio que deberíamos esperar en el conjunto completo de tweets. La tabla tiene que estar ordenada en orden ascendente de hora.

In [57]:

```
#sqlContext.sql("""SELECT hour(created_at) AS hour,
 #                date_format(date(created_at), "0M-d-YY") AS day,
  #               COUNT(_id ) as count  FROM tbl GROUP BY hour
   #              ORDER BY count DESC""")

sqlContext.sql("DROP TABLE IF EXISTS tweets_hour_day")
sqlContext.registerDataFrameAsTable(tweets_hour_day, "tweets_hour_day")

tweets_hour=sqlContext.sql("""SELECT hour,
                              100*AVG(count) AS tweets
                              FROM tweets_hour_day
                              GROUP BY hour
                              ORDER BY hour""")
#tweets_hour.limit(24).show()
```

In [58]:

```
assert tweets_hour.first().hour == 0 and round(tweets_hour.first().tweets) == 26189
```

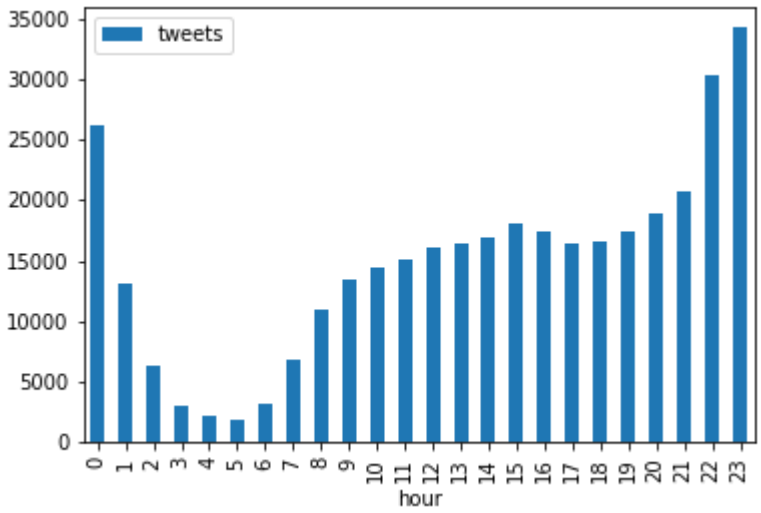
Por último, tenéis que producir un gráfico de barras utilizando Pandas (https://pandas.pydata.org/) donde se muestre la información que acabáis de generar. Primero transformad la tabla `tweets_hour` a pandas utilizando el método `toPandas()` . Plotead la tabla resultante utilizando la funcionalidad gráfica de pandas. (https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.plot.bar.html)

In [59]:

```
tweets_hour.toPandas().plot.bar(x='hour',y='tweets')
```

Out[59]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3a0fd22dd8>
```



Parte 3.2: Estratificado

En muchas ocasiones el sampling homogéneo no es adecuado ya que por la propia estructura de los datos determinados segmentos pueden estar sobrerrepresentadas. Este es el caso que observamos en los tweets donde las grandes áreas urbanas están sobrerepresentadas si lo comparamos con el volumen de población.

En esta actividad vamos a ver cómo aplicar esta técnica al dataset de tweets, para obtener un sampling que respete la proporción de diputados por provincia.

En España, el proceso electoral asigna un volumen de diputados a cada provincia que depende de la población y de un porcentaje mínimo asignado por ley. En el contexto Hive que hemos creado previamente (`hiveContext`) podemos encontrar una tabla (`province_28a`) que contiene información sobre las circunscripciones electorales. Cargad ésta tabla en una variable con nombre `province` .

In [60]:

```
#hiveContext.tables().show()
#tweets = hiveContext.table("default.tweets28a_sample25")
#tweets_place = hiveContext.sql("""SELECT user.screen_name, MAX(user.friends_count) AS frie
 #                          COUNT(user.screen_name) AS tweets, MAX(user.followers_count) AS
  #                         FROM tweets_sample WHERE user.lang == 'es'
   #                        GROUP BY user.screen_name ORDER BY tweets DESC""")
province=hiveContext.table("default.province_28a")
province.limit(20).show()
```

```
+-----------+-----------+-----------------+----------+---------+
|    capital|   province|              ccaa|population|diputados|
+-----------+-----------+-----------------+----------+---------+
|     Teruel|     Teruel|           Aragón|     35691|        3|
|      Soria|      Soria|  Castilla y León|     39112|        2|
|    Segovia|    Segovia|  Castilla y León|     51683|        3|
|     Huesca|     Huesca|           Aragón|     52463|        3|
|     Cuenca|     Cuenca|Castilla-La Mancha|     54898|        3|
|      Ávila|      Ávila|  Castilla y León|     57697|        3|
|     Zamora|     Zamora|  Castilla y León|     61827|        3|
|Ciudad Real|Ciudad Real|Castilla-La Mancha|     74743|        5|
|   Palencia|   Palencia|  Castilla y León|     78629|        3|
| Pontevedra| Pontevedra|          Galicia|     82802|        7|
|     Toledo|     Toledo|Castilla-La Mancha|     84282|        6|
|Guadalajara|Guadalajara|Castilla-La Mancha|     84910|        3|
|      Ceuta|      Ceuta|            Ceuta|     85144|        1|
|    Melilla|    Melilla|           Melilla|     86384|        1|
|    Cáceres|    Cáceres|      Extremadura|     96098|        4|
|       Lugo|       Lugo|           Galicia|     98025|        4|
|     Girona|     Girona|         Cataluña|    100266|        6|
|     Orense|     Orense|           Galicia|    105505|        4|
|       Jaén|       Jaén|        Andalucía|    113457|        5|
|      Cádiz|      Cádiz|        Andalucía|    116979|        9|
+-----------+-----------+-----------------+----------+---------+
```

In [61]:

```
assert province.count() == 52, "Incorrect answer"
```

Para hacer un sampling estratificado lo primero que tenemos que hacer es determinar la fracción que queremos asignar a cada categoría. En este caso queremos una fracción que haga que el ratio tweets diputado sea igual para todas las capitales de provincia. Tenemos que tener en cuenta que la precisión de la geolocalización en Twitter és normalmente a nivel de ciudad. Por eso, para evitar incrementar la complejidad del ejercicio, vamos a utilizar los tweets en capitales de provincia como proxy de los tweets en toda la provincia.

Lo primero que tenéis que hacer es crear un tabla `info_tweets_province` que debe contener:

- ***capital:*** nombre de la capital de provincia.
- ***tweets:*** número de tweets geolocalizados en cada capital
- ***diputados:*** diputados que asignados a la provincia.
- ***ratio_tweets_diputado:*** número de tweets por diputado.

Debéis ordenar la lista por `ratio_tweets_diputado` en orden ascendente.

Nota: Podéis realizar este ejercicio de muchas maneras, probablemente la más fácil es utilizar la tabla `tweets_place` que habéis generado en el apartado 2.2.1. Recordad cómo utilizar el `join()`

In [62]:

```
#tweets_place = hiveContext.sql("""SELECT place.name AS name, COUNT(place.name) AS tweets
 #                           FROM tweets28a_sample25 WHERE place.name!='null'
  #                          GROUP BY place.name ORDER BY tweets DESC""")
#tweets_place.limit(40).show()
#hiveContext.sql("SELECT place.name FROM tweets28a_sample25 WHERE place.name!='null'")

#retweeted = sqlContext.sql("""SELECT user_agg.screen_name, MAX(user_agg.friends_count) AS
 #                           MAX(user_agg.followers_count) AS followers_count,
  #                          MAX(user_agg.tweets) AS tweets,
   #                         COUNT(tweets_sample.retweeted_status) AS retweeted,
    #                        COUNT(tweets_sample.retweeted_status)/MAX(user_agg.tweets) AS
     #                       FROM tweets_sample
      #                      INNER JOIN user_agg
       #                     ON tweets_sample.retweeted_status.user.screen_name = user_agg.
        #                    GROUP BY user_agg.screen_name ORDER BY ratio_tweet_retweeted D

sqlContext.sql("DROP TABLE IF EXISTS tweets_place")
sqlContext.registerDataFrameAsTable(tweets_place, "tweets_place")

sqlContext.sql("DROP TABLE IF EXISTS province")
sqlContext.registerDataFrameAsTable(province, "province")

info_tweets_province=sqlContext.sql("""SELECT province.capital,
                MAX(tweets_place.tweets) AS tweets,
                MAX(province.diputados) AS diputados,
                MAX(tweets_place.tweets)/MAX(province.diputados) AS ratio_tweets_diputado
                FROM tweets_place
                INNER JOIN province
                ON tweets_place.name=province.capital
                GROUP BY province.capital
                ORDER BY ratio_tweets_diputado""")
info_tweets_province.limit(20).show()
```

```
+------------------+------+---------+--------------------+
|           capital|tweets|diputados|ratio_tweets_diputado|
+------------------+------+---------+--------------------+
|             Teruel|     8|        3|   2.6666666666666665|
|          Pontevedra|    29|        7|    4.142857142857143|
|             Zamora|    23|        3|    7.666666666666667|
|             Huesca|    26|        3|    8.666666666666666|
|            Segovia|    28|        3|    9.333333333333334|
|              Cádiz|   108|        9|                12.0|
|              Soria|    25|        2|                12.5|
|             Cuenca|    39|        3|                13.0|
|         Ciudad Real|    67|        5|                13.4|
|               Lugo|    56|        4|                14.0|
|           Pamplona|    77|        5|                15.4|
|               Jaén|    86|        5|                17.2|
|         Guadalajara|    56|        3|   18.666666666666668|
|             Cáceres|    75|        4|               18.75|
|Santa Cruz de Ten...|   140|        7|                20.0|
|             Toledo|   124|        6|   20.666666666666668|
|            Albacete|    83|        4|               20.75|
|       San Sebastián|   127|        6|   21.166666666666668|
|            Palencia|    67|        3|   22.333333333333332|
|           Tarragona|   135|        6|                22.5|
+------------------+------+---------+--------------------+
```

In [63]:

```
output = info_tweets_province.first()
maximum_ratio = floor(output.ratio_tweets_diputado * 100) / 100
assert output.capital == "Teruel" and output.tweets == 8 and output.diputados == 3, "Incorr
```

Lo primero que vamos a necesitar es un diccionario con nombre `ratios` donde cada capital de provincia es una llave y su valor asociado es la fracción de tweets que vamos a samplear. En este caso lo que queremos es que el ratio de tweets por cada diputado sea similar para cada capital de provincia.

Como queremos que el sampling sea lo más grande posible y no queremos que ninguna capital este infrarepresentada el ratio de tweets por diputado será el valor más pequeño podéis observar en la tabla `info_tweets_province`, que corresponde a 11.66 tweets por diputado en Teruel. Tenéis este valor guardado en la variable `maximum_ratio`.

Nota: El método `collectAsMap()` transforma un PairRDD en un diccionario.

In [64]:

```
ratios=info_tweets_province.rdd.map(lambda x:(x[0],x[2])).collectAsMap()
ratios['Albacete']
#info_tweets_province.rdd.map(lambda x:(x[0],100*x[2]/338)).take(50)
#sum(info_tweets_province.rdd.map(lambda x:x.tweets).take(60))
#info_tweets_province.rdd.map(lambda x:(x[0],x[2])).take(40)
#info_tweets_province.rdd.map(lambda x:x.tweets).take(50)
#maximum_ratio
maximum_ratio
```

Out[64]:

```
2.66
```

In [ ]:

```
assert ratios['Albacete'] == 0.12819277108433735, "Incorrect output"
```

Generad una tabla `geo_tweets` con todos los tweets geolocalizados.

In [ ]:

```
# YOUR CODE HERE
raise NotImplementedError()
```

Ahora ya estamos en disposición de hacer el sampling estratificado por población. Para ello podéis utilizar el método `sampleBy()`. Utilizad 42 como seed del generador pseudoaleatorio.

Esquema

```
seed = 42
sample = <FILL IN>
```

In [ ]:

```
# YOUR CODE HERE
raise NotImplementedError()
```

Para visualizar el resultado del sampling vais a crear una tabla `info_sample` que contenga la siguiente información:

- ***capital:*** nombre de la capital de provincia.
- ***tweets:*** número de tweets sampleados en cada capital
- ***diputados:*** diputados que asignados a la provincia.
- ***ratio_tweets_diputado:*** número de tweets por diputado.

Ordenad la tabla resultante por orden de `ratio_tweets_diputado` ascendente.

In [ ]:

```
# YOUR CODE HERE
raise NotImplementedError()

info_sample.limit(20).show()
```

In [ ]:

```
output = info_sample.first()
assert output.capital == "Melilla" and output.tweets == 1  and output.diputados == 1 and ou
```

Como veis el sampling no es exacto, es una aproximación. Pero como podéis imaginar acercar el sampling a la representatividad electoral de las regiones son necesarios en muchos análisis.

Para comprobarlo contad primero todos los hashtags presentes en la tabla `geo_tweets` tal como hemos hecho en el apartado 2.2.2 y ordenad el resultado por número de tweets en orden descendente. Guardad la tabla en la variable `hashtagsTable` .

In [ ]:

```
# YOUR CODE HERE
raise NotImplementedError()

hashtagsTable.limit(20).show()
```

In [ ]:

```
output = hashtagsTable.first()
assert output.hashtag == "#28A" and output.num >= 1700, "Incorrect answer"
```

Comparad este resultado con el que obtenemos cuando creamos una tabla `hashtagsTable_sample` donde contamos los hashtags en el sample estratificada. Ordenad la tabla por número de tweets en orden descendente.

In [ ]:

```
# YOUR CODE HERE
raise NotImplementedError()

hashtagsTable_sample.limit(20).show()
```

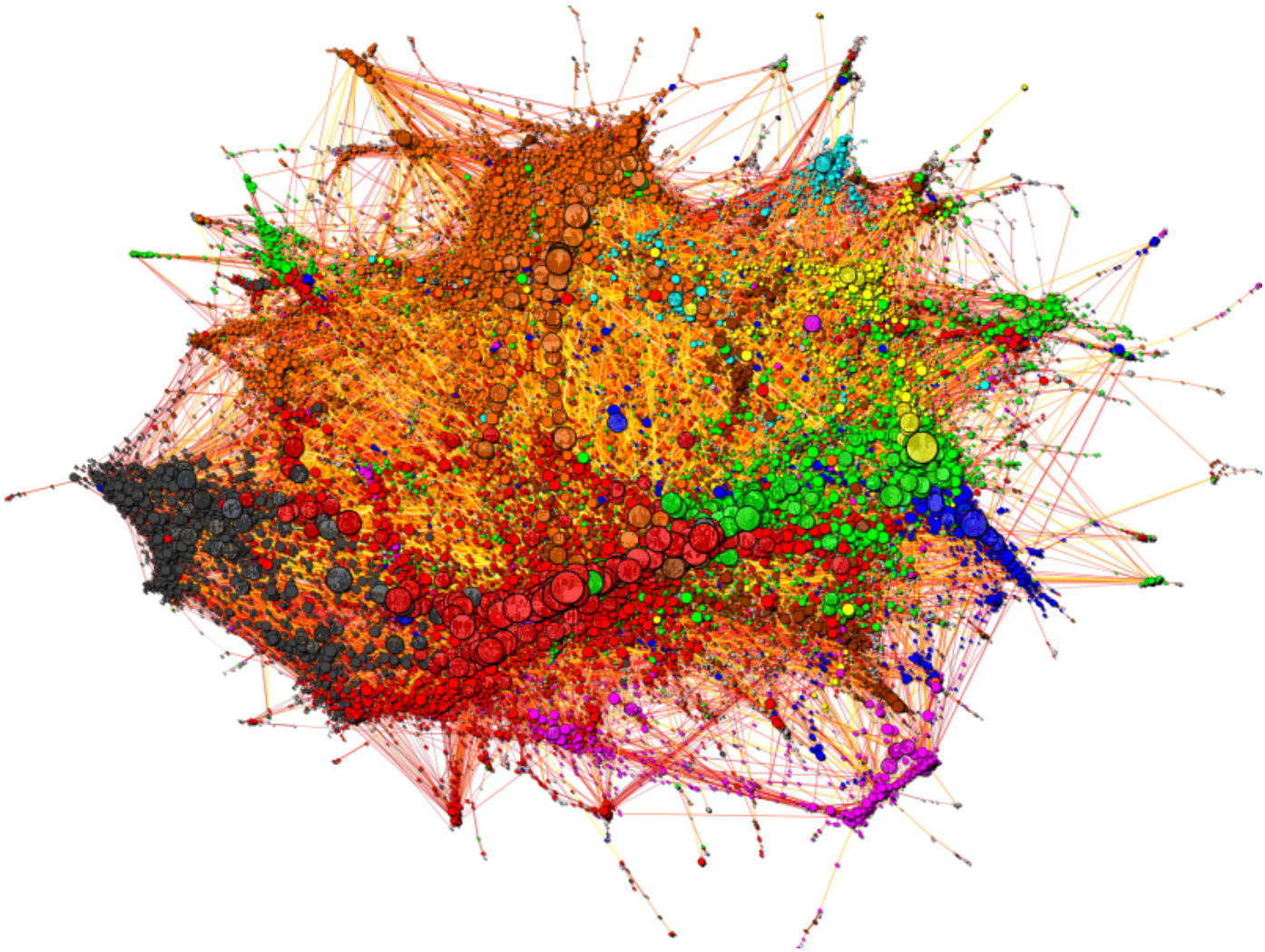
In [ ]:

```
output = hashtagsTable_sample.first()
assert output.hashtag == "#28A" and output.num >= 35, "Incorrect answer"
```

Parte 4: Introducción a los datos relacionales

El hecho de trabajar con una base de datos que contiene información generada en una red social nos permite introducir el concepto de datos relacionales. Podemos definir datos relacionales como aquellos en los que existen relaciones entre las entidades que constituyen la base de datos. Si estas relaciones son binarias, relaciones 1 a 1, podemos representar las relaciones como un grafo compuesto por un conjunto de vértices $\mathcal{V}$ y un conjunto de aristas $\mathcal{E}$ que los relacionan.

En el caso de grafos que emergen de manera orgánica, este tipo de estructura va más allá de los grafos regulares que seguramente conocéis. Este tipo de estructuras se conocen como redes complejas (https://es.wikipedia.org/wiki/Red_compleja). El estudio de la estructura y dinámicas de este tipo de redes ha contribuido a importantes resultados en campos tan dispares como la física, la sociología, la ecología o la medicina.



En esta última parte de la práctica vamos ha trabajar con este tipo de datos. En concreto vamos a modelar uno de los posibles relaciones presentes en el dataset, la red de retweets.

Parte 4.1: Generar la red de retweets

Parte 4.1.1: Construcción de la edgelist

Lo primero se os pide es que generéis la red. Hay diversas maneras de representar una red compleja, por ejemplo, si estuvierais interesados en trabajar en ellas desde el punto de vista teórico, la manera más habitual de representarlas es utilizando una matriz de adyacencia (https://es.wikipedia.org/wiki/Matriz_de_adyacencia). En esta práctica vamos a centrarnos en el aspecto computacional, una de las maneras de mas eficientes (computacionalmente hablando) de representar una red es mediante su *edge list* (https://en.wikipedia.org/wiki/Edge_list), una tabla que especifica la relación a parejas entre las entidades.

Las relaciones pueden ser bidireccionales o direccionales y tener algún peso asignado o no (weighted or unweighted). En el caso que nos ocupa, estamos hablando de una red dirigida, un usuario retuitea a otro, y podemos pensarla teniendo en cuenta cuántas veces esto ha pasado.

Lo primero que haréis para simplificar el cómputo, es crear un sample homogéneo sin reemplazo del 1% de los tweets. Utilizando los conocimientos que habéis aprendido en el apartado 3.1. Utilizaremos 42 como valor para la seed.

Esquema

```
seed = 42
sample = tweets.<FILL IN>
```

In [65]:

```
sample = tweets.sample(withReplacement=None,fraction=0.01,seed=42)
sample.take(2)
```

Out[65]:

```
[Row(_id='1117169844984610817', created_at=datetime.datetime(2019, 4, 13, 2
2, 57, 1), lang='es', place=None, retweeted_status=None, text='@Earco1977 Pu
es a mí me encantaría poder gobernar sin el PSOE.', user=Row(followers_count
=5893, friends_count=4783, id_str='509047038', lang='es', screen_name='Maria
_Beatle', statuses_count=15161)),
 Row(_id='1117169933849247744', created_at=datetime.datetime(2019, 4, 13, 2
2, 57, 22), lang='en', place=None, retweeted_status=Row(_id='111716531725446
7584', user=Row(followers_count=3916195, friends_count=1464, id_str='1382031
34', lang='en', screen_name='AOC', statuses_count=7699)), text='RT @AOC: “Th
ey’re a billion-dollar company because of us.\n\nWorking-class Americans hav
en’t been this fed up with their employers since the…', user=Row(followers_c
ount=105, friends_count=51, id_str='18158360', lang='en', screen_name='cspen
gler', statuses_count=20694))]
```

Type *Markdown* and LaTeX: α^2

Ahora vais a crear una tabla `edgelist` con la siguiente información:

- ***src:*** usuario que retuitea
- ***dst:*** usuario que es retuiteado
- ***weight:*** número de veces que un usuario retuitea a otro.

Filtrar el resultado para que contenga sólo las relaciones con un weight igual o mayor a dos.

In [66]:

```
edgelist=sample.rdd.filter(lambda x:x[4]!=None).map(lambda x:(x[6][4],x[4][1][4])).map(lamb
L = edgelist.count()

print("There are {0} edges on the network.".format(L))
edgelist.toDF().withColumnRenamed("_1","src")\
                        .withColumnRenamed("_2","dst")\
                        .withColumnRenamed("_3","weight").show(517)
```

```
There are 517 edges on the network.
+---------------+---------------+------+
|            src|            dst|weight|
+---------------+---------------+------+
|     nievesfouz|      populares|     2|
|cassandrasince1|  Santi_ABASCAL|     2|
|        kena264|     yoanferrer|     2|
|    Manudocalin| voxnoticias_es|     2|
|      el_partal|         mrenau|     2|
|      Sekandose|    CastigadorY|     2|
|Enedina37049621|           PSOE|     4|
|     kafkanario|   ahorapodemos|     3|
|         xurxox|       En_Marea|     2|
|  manuperez2002|   Mariagtriana|     2|
|     mirovira75|           KRLS|     2|
| CsSantaEularia|   CiudadanosCs|     4|
|       lmpg_twi|   ahorapodemos|     2|
|JuanAnt82323602|         vox_es|     2|
|       ssergi30|ldpsincomplejos|     2|
```

In [67]:

```
assert L == 517, "Incorrect ouput"
```

Parte 4.1.2: Centralidad de grado

Uno de los descriptores más comunes en el análisis de redes es el grado. El grado cuantifica cuántas aristas están conectadas a cada vértices. En el caso de redes dirigidas como la que acabamos de crear este descriptor está descompuesto en el:

- **in degree**: cuantas aristas apuntan al nodo
- **out degree**: cuantas aristas salen del nodo

Si haces un ranquing de estos valores vais a obtener una medida de centralidad, la centralidad de grado (https://en.wikipedia.org/wiki/Centrality#Degree_centrality), de cada uno de los nodos.

Se os pide que generéis una tabla `outDegree` con la información:

- ***screen_name:*** nombre del usuario.
- ***outDegree:*** out degree del nodo.

Ordenado la tabla por out degree en orden descendente.

In [68]:

```
sample = tweets.sample(withReplacement=None,fraction=0.01,seed=42)
#sample.take(5)
#sample.rdd.filter(lambda x:x[4]!=None).map(lambda x:x[6][4]).map(lambda x:(x,1)).reduceByK
#outDegree.limit(20).show()
#outDegree_aux=edgelist.map(lambda x:((x[0],x[2]),1)).reduceByKey(lambda a,b:a+b).map(lamba
#outDegree=outDegree_aux.toDF().withColumnRenamed("_1","screen_name")\
#                        .withColumnRenamed("_2","outDegree")
outDegree_aux=edgelist.map(lambda x:(x[0],1)).reduceByKey(lambda a,b:a+b).sortBy(lambda x:-
outDegree=outDegree_aux.toDF().withColumnRenamed("_1","screen_name")\
                       .withColumnRenamed("_2","outDegree")
outDegree.limit(20).show()
```

```
+---------------+---------+
|    screen_name|outDegree|
+---------------+---------+
|  antoniobb1953|        3|
|    Manudocalin|        3|
|     Nacho_JISF|        2|
|      Juancarfg|        2|
| ALFONSOLODEIRO|        2|
|     ArwenPlaza|        2|
|     aa21623129|        2|
|    RosaMar6254|        2|
|       cukianaa|        2|
|      ppoleiros|        2|
|Migueln53227148|        2|
|   merchi_otero|        2|
|    Karmaleonic|        2|
|      ArmGonGal|        2|
|     michiki_ta|        2|
|PisandoFuerte10|        2|
|      el_partal|        2|
|    mariasvilas|        2|
|  mariarossa004|        2|
|     carrasquem|        2|
+---------------+---------+
```

In [69]:

```
output = outDegree.first()
assert output.screen_name == "Manudocalin" and output.outDegree == 3, "Incorrect output"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-69-d7b9c597609a> in <module>
      1 output = outDegree.first()
----> 2 assert output.screen_name == "Manudocalin" and output.outDegree == 3
, "Incorrect output"

AssertionError: Incorrect output
```

Se os pide ahora que generéis una tabla `inDegree` con la información:

- ***screen_name:*** nombre del usuario.

- ***inDegree:*** in degree del nodo.

Ordenad la tabla por in degree en orden descendente.

In [70]:

```
sample = tweets.sample(withReplacement=None,fraction=0.01,seed=42)
#sample.rdd.filter(lambda x:x[4]!=None).map(lambda x:x[6][4]).map(lambda x:(x,1)).reduceByK
#sample.rdd.filter(lambda x:x[4]!=None).map(lambda x:(x[4][1][4],x[6][4])).map(lambda x:(x,
#sample.rdd.filter(lambda x:x[4]!=None).map(lambda x:(x[4][1][4],x[6][4])).map(lambda x:(x,

#inDegree.limit(20).show()

#outDegree_aux=edgelist.map(lambda x:((x[0],x[2]),1)).reduceByKey(lambda a,b:a+b).map(lamba
#outDegree=outDegree_aux.toDF().withColumnRenamed("_1","screen_name")\
#                        .withColumnRenamed("_2","outDegree")
#outDegree.limit(20).show()

inDegree_aux=edgelist.map(lambda x:(x[1],1)).reduceByKey(lambda a,b:a+b).sortBy(lambda x:-x
inDegree=inDegree_aux.toDF().withColumnRenamed("_1","screen_name")\
                        .withColumnRenamed("_2","inDegree")
inDegree.limit(20).show()
```

```
+---------------+--------+
|    screen_name|inDegree|
+---------------+--------+
|           PSOE|      45|
|   ahorapodemos|      37|
|         vox_es|      26|
|   CiudadanosCs|      26|
|      populares|      25|
|  Santi_ABASCAL|      18|
|Front_Republica|       9|
|      JuntsXCat|       9|
|  AlbanoDante76|       7|
|           KRLS|       6|
|sanchezcastejon|       6|
|   Esquerra_ERC|       5|
|    CastigadorY|       5|
|    protestona1|       5|
|      XoanXoann|       4|
| hermanntertsch|       4|
|     rcabrero75|       4|
|Pablo_Iglesias_|       4|
|       ivanedlm|       4|
|   pablocasado_|       4|
+---------------+--------+
```

In [71]:

```
output = inDegree.first()
assert output.screen_name == "PSOE" and output.inDegree == 45, "Incorrect output"
```

Part 4.2: Graphframes

Este tipo de estructuras es muy común en muchos datasets y su análisis cada vez se ha vuelto más habitual.

Para simplificar las operaciones y el análisis vamos a utilizar una librería específicamente diseñada para trabajar en redes en sistemas distribuidos: **Graphframes** (https://graphframes.github.io/graphframes/docs/_site/index.html).

In [72]:

```
import sys
pyfiles = str(sc.getConf().get(u'spark.submit.pyFiles')).split(',')
sys.path.extend(pyfiles)
from graphframes import *
```

Lo primero que vamos ha hacer es crear un objeto `GraphFrame` que contendrà toda la información de la red.

En un paso previo ya hemos creado la *edge list* ahora vamos a crear una lista con los vértices. Crear una tabla `vertices` que contenga una única columna `id` con los nombre de usuario de todos los vértices. Recordad que hay vértices que puede que solo tengan aristas incidentes y otros que puede que no tengan (tenéis que utilizar la información de ambas columnas de la `edgelist`). Recordad que la lista de vértices es un conjunto donde no puede haber repetición de identificadores.

In [73]:

```
col1=edgelist.map(lambda x:(x[0],1)).reduceByKey(lambda a,b:a+b)
col2=edgelist.map(lambda x:(x[1],1)).reduceByKey(lambda a,b:a+b)
col3=col1.map(lambda x:x[0])+col2.map(lambda x:x[0])
col4=col3.map(lambda x:(x,1)).reduceByKey(lambda a,b:a+b)
vertices=col4.toDF().drop("_2").withColumnRenamed("_1","id")
#sqlContext.sql("DROP TABLE IF EXISTS vertices")
#sqlContext.registerDataFrameAsTable(col4,"vertices")
#sqlContext.sql("SELECT * from vertices").show()

N = vertices.count()
print("There are {0} nodes on the network.".format(N))
```

```
There are 682 nodes on the network.
```

In [74]:

```
assert N == 682, 'Incorrect output'
```

Al igual que con las aristas, podéis asignar atributos a los vértices. Generad tabla `vertices_extended` combinando la tabla `vertices` haciendo un *inner join* por `id` con la tabla `user_info` guardada en el contexto `hiveContext`. Ordenad la tabla resultante por followers en orden descendente

In [76]:

```
hiveContext = HiveContext(sc)
#hiveContext.tables().show()
user_info = hiveContext.table("default.user_info")
#retweeted=users_agg_new.join(user_retweets, users_agg_new.screen_name==user_retweets.scree
 #           .select(users_agg_new.screen_name, users_agg_new.friends_count, users_agg_new.
  #          .orderBy("retweeted",ascending=False)
vertices_extended=vertices.join(user_info, vertices.id==user_info.id)\
                    .select(vertices.id,user_info.lang,user_info.tweets,\
                    user_info.total_tweets,user_info.following,user_info.followers).orderBy
vertices_extended.show()


#.orderBy("retweeted",ascending=False)
#.select(users_agg_new.screen_name, users_agg_new.friends_count, users_agg_new.followers_co
#vertices_extended.limit(20).show()
```

```
+---------------+----+------+------------+---------+---------+
|             id|lang|tweets|total_tweets|following|followers|
+---------------+----+------+------------+---------+---------+
|        el_pais|  es|   880|      478720|      756|  6904062|
|Pablo_Iglesias_|  es|   160|       20949|     2831|  2297175|
|   ahorapodemos|  es|   858|      100336|     1571|  1381944|
|  Albert_Rivera|  es|   122|       54980|     2581|  1127429|
|        agarzon|  es|   134|       58265|     1204|  1049193|
|sanchezcastejon|  es|   146|       25232|     6126|  1025272|
|     eldiarioes|  es|  1082|      196012|      467|   978114|
|       iescolar|  es|   266|       75294|     5574|   939790|
|     publico_es|  es|   742|      349214|     1471|   934593|
|           KRLS|  ca|   129|       18732|     4270|   754475|
|      populares|  es|   748|       82229|     4469|   708893|
|           PSOE|  es|  1085|       96635|    13644|   680626|
|    elperiodico|  es|   521|      399635|    18446|   613042|
|         iunida|  es|   586|       96033|    10255|   564099|
|   CiudadanosCs|  es|  1359|      130740|    92934|   524129|
|     MonederoJC|  es|    78|       13905|      998|   514714|
|  InesArrimadas|  es|   107|        3737|     1260|   502455|
|revistamongolia|  es|   240|      128199|     3381|   435101|
|           COPE|  es|   656|      169877|      161|   367094|
|   Esquerra_ERC|  ca|  1256|      111477|    99387|   363474|
+---------------+----+------+------------+---------+---------+
only showing top 20 rows
```

In [77]:

```
output = vertices_extended.first()
assert output.id == "el_pais" and output.lang == "es" and output.followers == 6904062
```

Una vez tenemos la edgelist y la lista de vértices estamos en disposición de instanciar un objecto `GraphFrame` (https://graphframes.github.io/graphframes/docs/_site/api/python/graphframes.html). Instanciad este objeto en la variable `network` .

In [82]:

```
#localVertices = [(1,"A"), (2,"B"), (3, "C")]
#localEdges = [(1,2,"love"), (2,1,"hate"), (2,3,"follow")]
#v = sqlContext.createDataFrame(vertices, "id")
#e = sqlContext.createDataFrame(edgelist, ["src", "dst", "weight"])
#g = GraphFrame(v, e)
#v=vertices
#e=edgelist.toDF()
#g = GraphFrame(v, e)

v = vertices
e = sqlContext.createDataFrame(edgelist, ['src', 'dst', 'weight'])
network = GraphFrame(v, e)
```

In [83]:

```
assert type(network) == GraphFrame, "Incorrect answer"
```

El objeto que acabais de crear tiene muchas atributos y métodos para el analisis de redes (comprobad el API) (https://graphframes.github.io/graphframes/docs/_site/api/python/graphframes.html). Se os pide que utilizéis el atributo `inDegrees` del objeto que acabáis de crear para, conjuntamente con la transformación `orderBy`, generar una tabla `inDegreeGraph` ordenada descendentemente por grado.

In [92]:

```
inDegreeGraph=network.inDegrees.orderBy("inDegree",ascending=False)
inDegreeGraph.limit(20).show()
```

```
+---------------+--------+
|             id|inDegree|
+---------------+--------+
|           PSOE|      45|
|   ahorapodemos|      37|
|   CiudadanosCs|      26|
|         vox_es|      26|
|      populares|      25|
|  Santi_ABASCAL|      18|
|Front_Republica|       9|
|      JuntsXCat|       9|
|  AlbanoDante76|       7|
|sanchezcastejon|       6|
|           KRLS|       6|
|    CastigadorY|       5|
|   Esquerra_ERC|       5|
|    protestona1|       5|
|     rcabrero75|       4|
|Pablo_Iglesias_|       4|
| AntonioMaestre|       4|
|socialistes_cat|       4|
|       marubimo|       4|
|       ivanedlm|       4|
+---------------+--------+
```

In [93]:

```
output = inDegreeGraph.first()
assert output.id == "PSOE" and output.inDegree == 45, "Incorrect answer"
```

Haced lo mismo con el atributo `outDegrees` para, conjuntamente con la transformación `orderBy`, generar una tabla `outDegreeGraph` que contenga la informació del out degree en orden descendente.

In [95]:

```
outDegreeGraph=network.outDegrees.orderBy("outDegree",ascending=False)
outDegreeGraph.limit(20).show()
```

```
+---------------+---------+
|             id|outDegree|
+---------------+---------+
|    Manudocalin|        3|
|  antoniobb1953|        3|
|     carrasquem|        2|
|     Nacho_JISF|        2|
|       cukianaa|        2|
|    Karmaleonic|        2|
|     aa21623129|        2|
| lanzarotejesus|        2|
| Maria_pilar_ga|        2|
|      Juancarfg|        2|
|      fatimar56|        2|
|     ArwenPlaza|        2|
|    RosaMar6254|        2|
| ALFONSOLODEIRO|        2|
|     mirovira75|        2|
|      ppoleiros|        2|
|Migueln53227148|        2|
|    mariaje1956|        2|
|  mariarossa004|        2|
|    mariasvilas|        2|
+---------------+---------+
```

In [96]:

```
output = outDegreeGraph.first()
assert output.id == "Manudocalin" and output.outDegree == 3, "Incorrect answer"
```

Parte 5: Preguntas teoricas y conceptuales (1 punto)

1. Explica las diferencias entre un SGBDR y Apache Hive. (0.3 puntos)

In [98]:

```
#¿Qué son?

#Un Sistema Gestor de Bases de Datos Relacionales es aquel sistema de bases de datos encarg
#datos de forma estructurada usando filas y columnas, siendo el resultado una tabla. Alguno
#y MySQL.
#Apache Hive es un sistema de almacenamiento del ecosistema hadoop que utiliza un lenguaje
#Language).

#Diferencias:

#No todos los SGBDR son distribuidos, Hive sí que lo es.
#Mientras que los SGBDR gestionan bases de datos, Hive gestiona almacenes de datos. Por lo
#no és un SGBDR. Cabe tener en cuenta que los SGBDR existían ya en la década de los 70, cua
#de grandes volúmenes de datos como los generados por las redes sociales no se planteaba. A
#respuesta a este problema, siendo más eficiente en estos escenarios.
#Una consecuencia de este punto es que el máximo tamaño de datos permitido por los SGBDR es
#Hive aplica el procedimiento "schema on read" para guardar datos, lo que significa que per
#una vez subidos sin necesidad de definir previamente su estructura.
#Los SGBDR aplican el procedimiento "schema on write", lo que significa que es necesario de
#antes de cargarlos y trabajar con ellos.
#El "sechama en read" es preferible en entornos big data en tanto en cuanto provee flexibil
#no estructurados o semiestructurados. Además hace más eficiente la subida de grandes volúm
#Hive aplica la idea "write once read many" lo que significa que los datos una vez guardado
#estudio de estos en profundidad.
#Los SGBDR están diseñados para aplicar el "read and write many times", priorizando así la
#profundo de datos.
#Los costes necesarios para el aumento de escalabilidad son mayores en los SGBDR.
```

2. Explica las relaciones que existe entre Apache Hive, su Metastore y HDFS. (0.3 puntos)

In [1]:

```
#El metastore de Hive es un servicio que permite almacenar metadatos de las tablas de Hive
#como MySQL o PostgreSQL.
#Estos metadatos incluyen información como el esquema de las tablas y su localización.
#Apache Hive por su parte permite consultar y analizar grandes volúmenes de datos que son a
#Hive guarda los metadatos en su metastore y no en HDFS para conseguir una baja latencia en
#de lectura y escritura son más largos que en el metastore.
```

3. Realiza una comparativa detallada (entre 5 y 10 lineas) entre Apache Hive e Impala. (0.4 puntos)

In [ ]:

```
#Los dos proporcionan motores de consultas basadas en SQL que pueden ser integrados en el e
#Hive sufre el llamado "cold start", lo que significa que las consultas son más lentas pues
#una tarea MapReduce que requiere un tiempo por parte de los nodos del cluster para ser lle
#no sucede porque no se invoca ninguna tarea MapReduce sino que utiliza procesos que se eje
#y que consultan directamente sobre HDFS. Las consultas son más rápidas en Impala.
#Hive es tolerante a fallos, si una consulta falla se encargará de que la parte de la consu
#reasignada y procesada de nuevo. En Impala, si la ejecución de una consulta falla, se debe
#Impala será por tanto más recomendable en escenarios donde prime la velocidad sobre la fia
#trabajos donde prime la robustez de ejecución frente a la velocidad.
```