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 [ ]:

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

PEC_5_2: Structured Streaming.

En esta PEC vamos a trabajar con Spark Structured Streaming (https://spark.apache.org/docs/2.2.0/structured-streaming-programming-guide.html) y su interacción con Kafka. Spark Structured Streaming es un motor de procesamiento de flujo escalable y tolerante a fallos construido sobre el motor Spark SQL.

Spark Structured Streaming nos permite realizar nuestro análisis de datos en streaming de la misma manera que lo hacemos con el procesamiento por lotes sobre datos estáticos. Ahora bien, hay que tener en cuenta que el structured streaming tiene una serie de ventajas. Por ejemplo, que el motor Spark SQL se encargará de ejecutar los análisis programados de forma incremental y continua, generando el resultado final a medida que los datos van entrando en el sistema. Structured Streaming se basa en la API de Dataset/DataFrame que se puede utilizar Scala, Java, Python o R para expresar agregaciones de transmisión, ventanas de tiempo de eventos, etc. Finalmente, el sistema asegura garantías de tolerancia a fallos de un extremo a otro a través de puntos de control y registros de escritura anticipada.

Los ejercicios están estructurados de en tres bloques para, de forma incremental, introduciros a la tecnología.

1. PARTE 1. Word Count con Structured Streaming (5 puntos)
2. PARTE 2. Operaciones de ventana sobre eventos temporales (4 puntos)
3. PARTE 3. Captura y procesamiento de datos en tiempo real de la API OpenSky (7 puntos)

IMPORTANTE: Para realizar esta práctica debes hacerlo mediante SSH desde terminal o VSCODE, y poner el código de la misma en este NOTEBOOK solo para su corrección.

Entrega:

Esta PEC se entregará mediante el NBGrader y el REC. El formato de entrega en el REC será un directorio comprimido en formato gnuzip bajo el nombre `PEC5_username.tar.gz` , substituyendo username por vuestro nombre de usuario. El contenido debe ser un fichero para cada programa Python por cada ejercicio indicando el apartado de los notebooks de enunciado, por ejemplo `PEC5_username_2_1_4.py` . Adjuntar los dos notebooks de la PEC, con el nombre `PEC5_1_username` , y `PEC5_2_username` con las salidas obtenidas que se piden, y las respuestas a las preguntas conceptuales planteadas.

PARTE 1. Word Count con Structured Streaming (5 puntos)

En esta primera parte de la PEC vamos a ver como implementar el típico word count con Structured Streaming insertando datos en el stream mediante un proceso netcat (https://en.wikipedia.org/wiki/Netcat) (que ya vimos en otras actividades). Como ya hicimos, el proceso netcat estará ejecutándose en una terminal vía SSH o VSCode y permitirá ir escribiendo palabras, que posteriormente van a ser contadas.

Para poder ejecutar este ejemplo necesitamos dos sesiones abiertas, en una vamos a escribir palabras que mediante netcat, datos que enviamos al puerto que tenéis asignado cada uno de vosotros, y en otra sesión vamos a ejecutar un programa mediante la API de Structured Streaming de Spark. Esta API nos permite procesar datos que se obtienen en streaming mediante Dataframes y con RDD como hemos visto en la PEC4.

Así pues, en una **sesión de terminal mediante SSH, no mediante Jupyter terminal** debéis ejecutar un netcat `$ nc -lk <puerto_asignado>` .

Ahora en otra sesión del intérprete (no en Jupyter notebook) vamos a ejecutar el siguiente programa en Python: `python3 PEC5_2_1_0.py localhost <PUERTO_ASIGNADO>` Para ello debéis crear el archivo "PEC5_2_1_0.py" y copiar el código siguiente dentro de él.

In [ ]:

```
"""
Programa que muestra exclusivamente las entradas via stream socket capturadas mediante Structured Streaming
 Para ponerlo en marcha utilizamos netcat
    `$ nc -lk <vuestro puerto>`
 y ejecutamos con
    `$ python3 PEC5_2_1_0.py localhost <vuestro puerto>`
"""
import sys
import findspark
findspark.init()

from pyspark import SparkConf, SparkContext, SQLContext, HiveContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

conf = SparkConf()
conf.setMaster("local[2]") #en local con dos threads
sc = SparkContext(conf=conf)

# Introducid el nombre de la app PEC5_ seguido de vuestro nombre de usuario


if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Uso: PEC5_2_1_1 <hostname> <port>", file=sys.stderr)
        exit(-1)

    host = sys.argv[1]
    port = int(sys.argv[2])

    spark = SparkSession\
        .builder\
        .appName(<PEC5_USUARIO>)\
        .getOrCreate()

    # Creamos el dataframe representando el stream de linias del netcat desde host:port
    lines = spark\
        .readStream\
        .format('socket')\
        .option('host', host)\
        .option('port', port)\
        .load()

    # Separamos las linias en palabras
    words = lines.select(
        explode(
            split(lines.value, ' ')
        ).alias('palabra')
    )

    wordCounts = words.groupBy('palabra').count()

    # Ejecutamos la consulta que muestra por consola el word count
    query = wordCounts\
        .writeStream\
        .outputMode('complete')\
        .format('console')\
        .start()
    #evitamos que el programa finalice mientras la consulta se ejecuta

    query.awaitTermination()
```

Explicación del código de ejemplo:

Para implementar un código con Structured Streaming, lo primero que necesitamos es una instancia de SparkSession. A las versiones modernas de Spark, la clase SparkSession es el punto de entrada de las aplicaciónes Spark para cualquier tipo de las APIs incluidas (RDD, SparkSQL, Streaming, etc.).

Mediante el objeto spark vamos a configurar una lectura de datos en streaming (propiedad readStream) reportados en el puerto que tenéis asociado, dado que es donde estará el netcat funcionando. Esta propiedad devuelve un objeto DataStreamReader que puede ser utilizado para leer flujos de datos en un DataFrame de streaming.

El DataFrame (`lines`) representa una tabla ilimitada que contiene la transmisión de datos de texto. Esta tabla contiene, por defecto, los datos en una columna de cadenas denominada `value` , y cada línea de los datos de texto de transmisión se convierte en una fila de la tabla. Tened en cuenta que aunque el objecto esté configurado todavía no está recibiendo ningún dato ya que solo estamos configurando la transformación y aún no hemos comenzado a recibir datos.

Seguidamente vamos a empezar a transformar los datos entrantes. Primeramente, separaremos pues las líneas en palabras, generando un nuevo DF (de forma similar a como lo hacíamos con la operación flatmap en RDDs). Para ello, en este caso, utilizaremos las funciones explode y split que están explicadas en https://spark.apache.org/docs/2.2.0/api/python/pyspark.sql.html (https://spark.apache.org/docs/2.2.0/api/python/pyspark.sql.html). Es importante fijarse en el detalle del cambio de nombre de la columna por defecto (`value`) a `palabra` , ya que a partir de ahora nos vamos a referir a ella en estos términos.

Ahora ya tendríamos una tabla con una columna llamada `palabra` , a la que nos podremos referir para realizar operaciones. Para completar el "wordcount" utilizamos la función `groupby(...)` combinada con `count(...)` de forma similar a lo que haríamos con SQL. La función count nos va a crear una nueva columna con los contajes como podréis ver en el resultado.

Una vez que hemos configurado la consulta (análisis) sobre los datos de transmisión, declaramos la consulta para comenzar a recibir los datos y contar las palabras. Para hacer esto, vamos a configurar la salida del análisis para que imprima el conjunto completo de recuentos, especificado por `outputMode("complete")` y configurado para mostrar por consola los datos cada vez que se actualizan. Finalmente iniciamos el cálculo de streaming usando `start()` .

Finalmente para evitar que el programa acabe vamos a utilizar la función `query.awaitTermination()` que va a bloquear el programa hasta que se pulse `Ctrl+C` .

Salida de ejemplo:

```
---------------
Batch: 1
---------------
+-------+-----+
|palabra|count|
+-------+-----+
|  tots!|    1|
|    dia|    1|
|    Bon|    1|
|      a|    1|
+-------+-----+
```

A partir de este ejemplo que hemos visto y que el alumno debe ejecutar para probar su funcionamiento, se pide:

> **Pregunta 1. (1 punto)** Realiza un programa en Python que cuente las palabras que empiezan por A y que tengan más de 3 counts. Debéis ejecutarlo el programa en una **terminal**, no dentro del Jupyter, y en otra terminal el netcat. Para ello utilizamos un programa Python en local mediante `./python3 PEC5_2_1_1.py localhost <puerto_asignado>`

Adjunta el código y la salida obtenida en **forma textual**.

NOTA: Se recomienda el uso de las operaciones Spark SQL (https://spark.apache.org/docs/2.2.0/api/python/pyspark.sql.html): filter(...), groupBy(...) y count(...) en combinación con las otras que creáis necesarias.

Salida de ejemplo:

```
 +-------+-----+
|palabra|count|
+-------+-----+
| Albert|    6|
+-------+-----+
```

In [ ]:

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

from pyspark import SparkConf, SparkContext, SQLContext, HiveContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

conf = SparkConf()
conf.setMaster("local[2]") #En local con dos threads
sc = SparkContext(conf=conf)

#Introducid el nombre de la app PEC5_ seguido de vuestro nombre de usuario


if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Uso: PEC5_2_1_1 localhost 20216", file=sys.stderr)
        exit(-1)

    host = sys.argv[1]
    port = int(sys.argv[2])

    spark = SparkSession\
        .builder\
        .appName('PEC5_vripollr')\
        .getOrCreate()

    #Creamos el dataframe representando el stream de linias del netcat desde host:port
    lines = spark\
        .readStream\
        .format('socket')\
        .option('host', host)\
        .option('port', port)\
        .load()

    #Separamos las linias en palabras
    words = lines.select(
        explode(
            split(lines.value, ' ')
        ).alias('palabra')
    )

    wordCounts = words.filter(words.palabra.startswith('A')).filter(length(words.palabra)>=3).groupBy('palabra').count()

    #Ejecutamos la consulta que muestra por consola el word count
    query = wordCounts\
        .writeStream\
        .outputMode('complete')\
        .format('console')\
        .start()
    #Evitamos que el programa finalice mientras la consulta se ejecuta

    query.awaitTermination()
```

Copia la salida obtenida en formato de texto:

In [ ]:

```
-------------------------------------------
Batch: 1
-------------------------------------------
+-------+-----+
|palabra|count|
+-------+-----+
|   AAAA|    1|
|    AAA|    1|
+-------+-----+
```

Ahora vamos a realizar un ejercicio que nos permita realizar una consulta SQL sobre los datos recibidos.

> **Pregunta 2. (1 punto)** Crea una tabla temporal para poder realizar una consulta SQL sobre las palabras que estamos obteniendo mediante streaming. El programa debe extraer las diferentes palabras de una frase y solo mostrar por consola aquellas que tengan una longitud superior a 3 caracteres.

NOTA 1: se recomienda revisar la opción "includeTimestamp" en la configuración del stream de datos y verificar que se está recogiendo la columna correspondiente al timestamp. Podéis usar la instrucción "linesDF.select('*')" para verificar qué os está llegando en el stream de datos.

NOTA 2: se recomienda el uso de las funciones "createOrReplaceTempView", el uso de las consultas "spark.sql".

Salida de ejemplo:

```
 +-------+--------------------+
|palabra|              tiempo|
+-------+--------------------+
|   Data|2021-12-2  12:21:...|
+-------+--------------------+
```

In [ ]:

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

from pyspark import SparkConf, SparkContext, SQLContext, HiveContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

conf = SparkConf()
conf.setMaster("local[2]") #En local con dos threads
sc = SparkContext(conf=conf)

#Introducid el nombre de la app PEC5_ seguido de vuestro nombre de usuario


if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Uso: PEC5_2_1_1 localhost 20216", file=sys.stderr)
        exit(-1)

    host = sys.argv[1]
    port = int(sys.argv[2])

    spark = SparkSession\
        .builder\
        .appName('PEC5_vripollr')\
        .getOrCreate()

    #Creamos el dataframe representando el stream de linias del netcat desde host:port
    lines = spark\
        .readStream\
        .format('socket')\
        .option('host', host)\
        .option('port', port)\
        .option('includeTimestamp', 'true')\
        .load()

    #Separamos las linias en palabras, seleccionamos su timestamp y nos quedamos con las que tienen length mayor que 3
    words_1 = lines.select(explode(split(lines.value, ' ')).alias('palabra'),lines.timestamp.alias('tiempo'))
    words_2 = words_1.filter(length(words_1.palabra)>=3)

    #Ejecutamos consulta
    query = words_2\
        .writeStream\
        .outputMode('append')\
        .format('console')\
        .option('truncate', 'false')\
        .start()
    #Evitamos que el programa finalice mientras la consulta se ejecuta

    query.awaitTermination()
```

Copia la salida obtenida en formato de texto:

In [ ]:

```
-------------------------------------------
Batch: 1
-------------------------------------------
+-------+----------------------+
|palabra|tiempo                |
+-------+----------------------+
|aaaa   |2022-12-30 21:44:05.543|
+-------+----------------------+
```

Pregunta 3. (1 punto) Modifica el programa para que haga uso del outputMode *append*. Debemos de guardar cada entrada en un fichero de texto en HDFS. Adjunta la salida del HDFS del contenido del directorio creado.

NOTA 1: Revisar en la documentación los output sinks (https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#output-sinks) que ofrece Structured Streaming. En este caso nosotros vamos a usar el "text" que se usa de forma equivalente al "parquet" documentado, junto con la configuración del path de destino como se observa en la documentación oficial enlazada.

Salida de ejemplo:

```
hdfs dfs -ls /user/<usuario>/pec5_1_3
Found 4 items
drwxr-xr-x   - usuario usuario          0 2021-12-02 12:43 /user/<usuario>/pec5_1_3/_spark_metadata
-rw-r--r--   3 usuario usuario          9 2021-12-02 12:43 /user/<usuario>/pec5_1_3/part-00000-499014ff-cf00-4f2f-a8a4-d2
82cdac1a19-c000.txt
-rw-r--r--   3 usuario usuario          7 2021-12-02 12:43 /user/<usuario>/pec5_1_3/part-00000-7d8cb984-95ad-4e50-8887-a7
2f4d2814a2-c000.txt
-rw-r--r--   3 usuario usuario          0 2021-12-02 12:43 /user/<usuario>/pec5_1_3/part-00000-a4918ce0-c930-439f-a5cd-a1
bd777f609b-c000.txt
```

In [ ]:

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

from pyspark import SparkConf, SparkContext, SQLContext, HiveContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

conf = SparkConf()
conf.setMaster("local[2]") #En local con dos threads
sc = SparkContext(conf=conf)

#Introducid el nombre de la app PEC5_ seguido de vuestro nombre de usuario


if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Uso: PEC5_2_1_1 localhost 20216", file=sys.stderr)
        exit(-1)

    host = sys.argv[1]
    port = int(sys.argv[2])

    spark = SparkSession\
        .builder\
        .appName('PEC5_vripollr')\
        .getOrCreate()

    #Creamos el dataframe representando el stream de linias del netcat desde host:port
    lines = spark\
        .readStream\
        .format('socket')\
        .option('host', host)\
        .option('port', port)\
        .option('includeTimestamp', 'true')\
        .load()

    #Separamos las linias en palabras, seleccionamos su timestamp y nos quedamos con las que tienen length mayor que 3
    words_1 = lines.select(explode(split(lines.value, ' ')).alias('palabra'),lines.timestamp.alias('tiempo'))
    words_2 = words_1.filter(length(words_1.palabra)>=3)
    #words.printSchema()

    #La opción .format('text') solo permite guardar una columna así que escogemos previamente una
    words_2.createOrReplaceTempView("Tabla")
    words_2_sql = spark.sql("select palabra from Tabla")

    #Ejecutamos consulta
    query = words_2_sql\
        .writeStream\
        .outputMode('append')\
        .format('text')\
        .option("path", "hdfs://Cloudera01/user/vripollr/prueba1")\
        .option("checkpointLocation", "checkpoint1")\
        .start()
    #Evitamos que el programa finalice mientras la consulta se ejecuta

    query.awaitTermination()
```

Copia la salida obtenida en el HDFS en formato de texto:

In [ ]:

```
vripollr@Cloudera01:~/sgraul/PEC5$ hdfs dfs -ls prueba1
Found 5 items
drwxr-xr-x   - vripollr vripollr          0 2022-12-31 12:54 prueba1/_spark_metadata
-rw-r--r--   3 vripollr vripollr          4 2022-12-31 12:54 prueba1/part-00000-28c002c4-daa1-4edf-9d44-ebc663334eba-c000.txt
-rw-r--r--   3 vripollr vripollr          6 2022-12-31 12:54 prueba1/part-00000-32fd0e58-4667-425e-880b-b13bc37f4c38-c000.txt
-rw-r--r--   3 vripollr vripollr          0 2022-12-31 12:54 prueba1/part-00000-5865bbe8-b002-4084-9345-e5c383c8152f-c000.txt
-rw-r--r--   3 vripollr vripollr          7 2022-12-31 12:54 prueba1/part-00000-d94bd42d-283b-48e6-80e8-459ae3dffee0-c000.txt
```

Pregunta 4.(1 punto) Realiza un programa en Python para que haga uso del outputMode update, y que muestre los datos entrantes por consola. La lectura debe realizar en intervalos de 5 segundos

Nota se recomienda el uso de la operación trigger (https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#triggers) en la configuración de la salida de resultados.

Salida de ejemplo:

```
-------------------------------------------
Batch: 5
-------------------------------------------
+-------+
|palabra|
+-------+
|    Big|
|   Data|
| Hadoop|
+-------+

-------------------------------------------
Batch: 6
-------------------------------------------
+-------+
|palabra|
+-------+
|  Spark|
+-------+
```

In [ ]:

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

from pyspark import SparkConf, SparkContext, SQLContext, HiveContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

conf = SparkConf()
conf.setMaster("local[2]") #En local con dos threads
sc = SparkContext(conf=conf)

#Introducid el nombre de la app PEC5_ seguido de vuestro nombre de usuario


if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Uso: PEC5_2_1_1 localhost 20216", file=sys.stderr)
        exit(-1)

    host = sys.argv[1]
    port = int(sys.argv[2])

    spark = SparkSession\
        .builder\
        .appName('PEC5_vripollr')\
        .getOrCreate()

    #Creamos el dataframe representando el stream de linias del netcat desde host:port
    lines = spark\
        .readStream\
        .format('socket')\
        .option('host', host)\
        .option('port', port)\
        .option('includeTimestamp', 'true')\
        .load()

    #Separamos las linias en palabras, seleccionamos su timestamp y nos quedamos con las que tienen length mayor que 3
    words_1 = lines.select(explode(split(lines.value, ' ')).alias('palabra'),lines.timestamp.alias('tiempo'))
    words_2 = words_1.filter(length(words_1.palabra)>=3)
    #words.printSchema()

    #Seleccionamos la columna palabra
    words_2.createOrReplaceTempView("Tabla")
    words_2_sql = spark.sql("select palabra from Tabla")

    #Ejecutamos consulta, la lectura se fija en 5 segundos
    query = words_2_sql\
        .writeStream\
        .outputMode('update')\
        .trigger(processingTime="5 second")\
        .format('console')\
        .start()

    #Evitamos que el programa finalice mientras la consulta se ejecuta

    query.awaitTermination()
```

Copia la salida obtenida en formato de texto:

In [ ]:

```
-------------------------------------------
Batch: 1
-------------------------------------------
+-------+
|palabra|
+-------+
+-------+

-------------------------------------------
Batch: 2
-------------------------------------------
+-------+
|palabra|
+-------+
|    aaa|
|aaaaaaa|
|  aaaaa|
+-------+

-------------------------------------------
Batch: 3
-------------------------------------------
+---------+
|  palabra|
+---------+
|aaaaaaaaa|
+---------+

-------------------------------------------
Batch: 4
-------------------------------------------
+-------+
|palabra|
+-------+
|    aaa|
+-------+
```

Pregunta 5.(1 punto) Explica las diferencias y similitudes entre los tipos de salidas existentes en Structured Streaming (complete, update y append). El texto debe ser claro, explicativo y tener una extensión de 10 líneas aproximadamente.

In [ ]:

```
#La diferencia principal entre los tres tipos de salida es la información que se muestra en la tabla. En el modo complete
#se muestra la tabla completa con todos los eventos recibidos, es decir, van acumulándose. En el modo append solo se muestran
#las filas que han sido agregadas a partir de un determinado trigger. Este trigger puede ser definido en la query mediante
#.trigger(). En el modo update solo se muestran en la tabla las filas que han sido actualizadas desde el último trigger.
#Como vemos, el modo append y el update son muy parecidos, en uno se muestran filas nuevas, en el otro las últimas actualizadas.
#Esta diferencia hace que el modo append sea aplicable solo en casos en los que las filas convenga que sean mostradas solo
#una vez, dejando el modo update para aquellas situaciones en que los valores de las filas estén cambiando continuamente
#y se necesite hacer una consulta de estos valores. Otra diferencia sería el tipo de queries que pueden ser aplicables, el
#modo append por lo general no es una buena opción para realizar agregaciones ya que este tipo de acciones requieren la
#consulta de datos anteriores.
#
#[1] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#output-modes
#[2] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html
```

PARTE 2. Operaciones de ventana sobre eventos temporales (4 puntos)

En esta parte vamos a trabajar con operaciones de ventana sobre eventos temporales. Para ello vamos a utilizar el formato *rate*. El source RateStreamSource (https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#api-using-datasets-and-dataframes) es una fuente de transmisión que genera números consecutivos con marca de tiempo y es utilizada habitualmente para hacer pruebas y PoC (*Proof of Concept*). Para configurar un RateStreamSource utilizaremos `format('rate')` , y el esquema de los datos entrantes es el adjunto siguiente. A diferencia de los ejercicios de la parte 1 no tendremos dos programas corriendo simultáneamente en el terminal, solo tendremos una, la de nuestro programa pyspark, dado que el formato rate se controla directamente desde la configuración del source Spark.

```
root
 |-- timestamp: timestamp (nullable = true)
 |-- value: long (nullable = true)
```

Pregunta 1. (1 punto) Realiza un programa mediante Structured Streaming que genere números mediante un formato *rate* como origen del streaming y donde debéis mostrar los que han aparecido dentro de cada ventana temporal ordenados por valor. Los números deben generarse cada segundo, y debemos utilizar una ventana de agrupación del streaming de 10 segundos y que se actualice cada 5 segundos.

NOTA 1 Revisar los input sources (https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#input-sources) y en particular el del "format('rate')" y sus opciones.

NOTA 2 Revisar el uso de las ventanas temporales (https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#input-sources).

In [ ]:

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

from pyspark import SparkConf, SparkContext, SQLContext, HiveContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

conf = SparkConf()
conf.setMaster("local[2]") #En local con dos threads
sc = SparkContext(conf=conf)

#Introducid el nombre de la app PEC5_ seguido de vuestro nombre de usuario

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Uso: PEC5_2_1_1 localhost 20216", file=sys.stderr)
        exit(-1)

    host = sys.argv[1]
    port = int(sys.argv[2])

    spark = SparkSession\
        .builder\
        .appName('PEC5_vripollr')\
        .getOrCreate()

    #Creamos el dataframe representando el stream de linias del netcat desde host:port. Con rowsPerSecond 1 indicamos que
    #se generen filas cada segundo
    lines = spark.readStream.format("rate").option("rowsPerSecond", 1).load()


    #Separamos las linias en palabras
    #words = lines.select(lines.value,lines.timestamp)

    #Agrupamos por ventanas
    windowedCounts = lines.groupBy(window(lines.timestamp, "10 seconds", "5 seconds"),lines.value).count().drop('count').orderBy(

    #Ejecutamos consulta
    query = windowedCounts\
        .writeStream\
        .outputMode('complete')\
        .format('console')\
        .option('truncate', 'false')\
        .start()
    #Evitamos que el programa finalice mientras la consulta se ejecuta

    query.awaitTermination()
```

Pregunta 2. (1 punto) Comenta el código, muestra la salida que has obtenido y coméntala en una extensión en 4 y 8 líneas. Haz especial énfasis en explicar por qué cada valor sale varias veces en los counts, si solamente se ha generado una vez mediante el format 'rate'.

Salida de ejemplo:

```
-------------------------------------------
Batch: 1
-------------------------------------------
+------------------------------------------+-----+
|window                                    |value|
+------------------------------------------+-----+
|[2022-12-21 22:06:10, 2022-12-21 22:06:20]|0    |
|[2022-12-21 22:06:05, 2022-12-21 22:06:15]|0    |
|[2022-12-21 22:06:10, 2022-12-21 22:06:20]|1    |
|[2022-12-21 22:06:15, 2022-12-21 22:06:25]|1    |
|[2022-12-21 22:06:15, 2022-12-21 22:06:25]|2    |
|[2022-12-21 22:06:10, 2022-12-21 22:06:20]|2    |
|[2022-12-21 22:06:15, 2022-12-21 22:06:25]|3    |
|[2022-12-21 22:06:10, 2022-12-21 22:06:20]|3    |
|[2022-12-21 22:06:15, 2022-12-21 22:06:25]|4    |
```

In [ ]:

```
-------------------------------------------
Batch: 1
-------------------------------------------
+-----------------------------------------+-----+
|window                                   |value|
+-----------------------------------------+-----+
|[2022-12-31 15:57:15, 2022-12-31 15:57:25]|0    |
|[2022-12-31 15:57:10, 2022-12-31 15:57:20]|0    |
|[2022-12-31 15:57:15, 2022-12-31 15:57:25]|1    |
|[2022-12-31 15:57:10, 2022-12-31 15:57:20]|1    |
|[2022-12-31 15:57:15, 2022-12-31 15:57:25]|2    |
|[2022-12-31 15:57:10, 2022-12-31 15:57:20]|2    |
|[2022-12-31 15:57:20, 2022-12-31 15:57:30]|3    |
|[2022-12-31 15:57:15, 2022-12-31 15:57:25]|3    |
|[2022-12-31 15:57:15, 2022-12-31 15:57:25]|4    |
|[2022-12-31 15:57:20, 2022-12-31 15:57:30]|4    |
|[2022-12-31 15:57:20, 2022-12-31 15:57:30]|5    |
|[2022-12-31 15:57:15, 2022-12-31 15:57:25]|5    |
|[2022-12-31 15:57:15, 2022-12-31 15:57:25]|6    |
|[2022-12-31 15:57:20, 2022-12-31 15:57:30]|6    |
|[2022-12-31 15:57:15, 2022-12-31 15:57:25]|7    |
|[2022-12-31 15:57:20, 2022-12-31 15:57:30]|7    |
|[2022-12-31 15:57:25, 2022-12-31 15:57:35]|8    |
|[2022-12-31 15:57:20, 2022-12-31 15:57:30]|8    |
|[2022-12-31 15:57:20, 2022-12-31 15:57:30]|9    |
|[2022-12-31 15:57:25, 2022-12-31 15:57:35]|9    |
+-----------------------------------------+-----+

#Cuando hemos agrupado por ventanas mediante el código window(lines.timestamp, "10 seconds", "5 seconds"), con el primer
#parámetro temporal hemos indicado que el windowLength es 10 segundos y con el segundo, que el slideInterval es 5 segundos.
#Lo que esto significa es que las ventanas para recoger los datos generados tienen una duración de 10 segundos y se desplazan
#de 5 en 5 segundos. Es decir, si lanzáramos el programa a las 00:00, la primera ventana se abriría en 00:00 - 00:10,
#la siguiente en 00:05 - 00:15, etc. Observemos lo que ocurre con el value 0 por ejemplo, si bien se ha generado una única vez,
#se ha recogido en dos ventanas distintas, en la ventana 15:57:10 - 15:57:20 y en la ventana 15:57:15 - 15:57:25, por eso
#aparece varias veces en los counts.
#Notemos que esto implica que el value 0 se ha generado en el intervalo 15:57:15 - 15:57:20.
```

En la ejecución de las consultas es muy interesante poder ir obteniendo información sobre el progreso realizado en el último disparador del flujo: qué datos se procesaron, cuáles fueron las tasas de procesamiento, latencias, etc.

> **Pregunta 3 (1 punto).** Modifica el programa para que muestre 3 métricas (https://spark.apache.org/docs/2.2.0/structured-streaming-programming-guide.html#monitoring-streaming-queries) del streaming mientras este se realiza. Solo se deben mostrar métricas mientras la consulta está activa.

In [ ]:

```
import time
import sys
import findspark
findspark.init()

from pyspark import SparkConf, SparkContext, SQLContext, HiveContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

conf = SparkConf()
conf.setMaster("local[2]") #En local con dos threads
sc = SparkContext(conf=conf)

#Introducid el nombre de la app PEC5_ seguido de vuestro nombre de usuario

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Uso: PEC5_2_1_1 localhost 20216", file=sys.stderr)
        exit(-1)

    host = sys.argv[1]
    port = int(sys.argv[2])

    spark = SparkSession\
        .builder\
        .appName('PEC5_vripollr')\
        .getOrCreate()

    #Creamos el dataframe representando el stream de linias del netcat desde host:port
    lines = spark.readStream.format("rate").option("rowsPerSecond", 1).load()


    #Separamos las linias en palabras
    #words = lines.select(lines.value,lines.timestamp)

    #Agrupamos por ventanas
    windowedCounts = lines.groupBy(window(lines.timestamp, "10 seconds", "5 seconds"),lines.value).count().drop('count').orderBy(

    #Ejecutamos consulta
    query = windowedCounts\
        .writeStream\
        .outputMode('complete')\
        .format('console')\
        .option('truncate', 'false')\
        .start()

    #El siguiente while evita que awaitTermination() bloquee el print de métricas a partir del batch 0
    while query.isActive:
        print('\n')
        print('-------------STATUS-------------')
        print('\n')
        print(query.status)
        print('\n')
        print('---------RECENT PROGRESS--------')
        print('\n')
        print(query.recentProgress)
        print('\n')
        print('----------LAST PROGRESS---------')
        print('\n')
        print(query.lastProgress)
        print('\n')
        time.sleep(10)
        #Añadimos un delay de 10 segundos para sincronizar los queries con las métricas

    #Evitamos que el programa finalice mientras la consulta se ejecuta

    query.awaitTermination()
```

Pregunta 4 (1 punto). Explica las 3 métricas aplicadas con una extensión entre 5 y 10 líneas propias.

Salida de ejemplo:

{'isDataAvailable': False, 'isTriggerActive': False, 'message': 'Initializing sources'}
{'stateOperators': [{'customMetrics': {'loadedMapCacheHitCount': 0, 'stateOnCurrentVersionSizeBytes': 25198,
'loadedMapCacheMissCount': 0}, 'numRowsUpdated': 0, 'memoryUsedBytes': 82798, 'numRowsTotal': 0}], 'timestamp': '2021-12-
03T10:16:01.657Z', 'sources': [{'description': 'RateStreamV2[rowsPerSecond=1, rampUpTimeSeconds=0, numPartitions=default',

```
'endOffset': 0, 'startOffset': None, 'processedRowsPerSecond': 0.0, 'numInputRows': 0}], 'runId': '9134994e-c17f-4277-974a-
21cd09cf9aea', 'durationMs': {'triggerExecution': 36450, 'walCommit': 48, 'getB[Stage 8:======>        (22 + 2) / 200]
```

In [ ]:

```
#Salida obtenida antes del batch 3:

-------------STATUS-------------


{'message': 'Processing new data', 'isDataAvailable': True, 'isTriggerActive': True}


---------RECENT PROGRESS--------


[{'id': '53e551e2-8159-4bb9-904e-9a4131695f05', 'numInputRows': 0, 'sink': {'description': 'org.apache.spark.sql.execution.stream


----------LAST PROGRESS---------


{'id': '53e551e2-8159-4bb9-904e-9a4131695f05', 'numInputRows': 10, 'stateOperators': [{'memoryUsedBytes': 146237, 'numRowsUpdated

#Las métricas utilizadas han sido query.status, query.recentProgress y query.lastProgress. Status da información sobre el estado
#actual de la query; así, se observa que antes del batch 3 se están procesando nuevos datos ('processing new data'), que existen
#datos disponibles para ser procesados ('isDataAvailable') y que existe un trigger activo ('isTriggerActive'). Recordemos que
#cuando hemos definido la ventana hemos definido el trigger slideInterval = 5 segundos. Recent progress da información acumulada
#sobre las actualizaciones más recientes del query mientras que last progress, la misma información pero solo de la última actual
#Se incluye diversa información como el 'id', un identificador del query que se mantiene en toda la ejecución (se puede comprobar
#se mantiene en recent progress); 'numInputRows', el número de filas de datos añadidas desde los sources (1 en este caso) en los
#definidos por los triggers, observemos que este parámetro toma los valores 0, 10, 15 en los primeros batches y se estabiliza en
#de los siguientes batches, este hecho concuerda con los parámetros introducidos en la implementación de las ventanas. El parámet
#'processedRowsPerSecond' toma valores cercanos a 1 lo cual concuerda también con los parámetros implementados ya que el stream l
#1 row/segundo (.option("rowsPerSecond", 1)). Se incluye otra información como la memoria utilizada, el 'timestamp' de la actuali
```

PARTE 3. Captura y procesamiento de datos en tiempo real mediante Kafka y Spark de datos obtenidos con la API OpenSky (7 puntos)

Es esta parte de la práctica vamos a trabajar la adquisición de datos en tiempo real de OpenSky (https://opensky-network.org/). OpenSky Network es una asociación sin ánimo de lucro con sede en Suiza que brinda acceso abierto a los datos de control de seguimiento de vuelos. Fue creado como un proyecto de investigación por varias universidades y entidades gubernamentales con el objetivo de mejorar la seguridad, confiabilidad y eficiencia del espacio aéreo. Su función principal es recopilar, procesar y almacenar datos de control de tráfico aéreo y proporcionar acceso abierto a estos datos al público. Esencialmente los datos de los aviones se obtienen vía satélite haciendo uso de Automatic Dependent Surveillance–Broadcast (ADS–B). Para realizar este ejercicio no es necesario registrarse en el sistema OpenSky dado que vamos a realizar actualizaciones de la información e vuelo sobre la superficie de España cada 10 segundos. La API está disponible este enlace (https://openskynetwork.github.io/opensky-api/python.html). El parámetro bbox es una tupla que indica la latitud mínima, máxima, y las longitudes mínimas y máximas.

Primeramente, vamos a utilizar el servicio OpenSkyApi para leer un rectángulo con las latitudes y longitudes que engloban la península ibérica.

Para ello debéis instalar (https://github.com/openskynetwork/opensky-api) la biblioteca en vuestro directorio del servidor Cloudera

1. Descargar en formato .zip el repositorio
2. Subir a vuestro directorio personal del servidor de Cloudera el zip.
3. Descomprimirlo.
4. Dentro del directorio que ha creador ejecutar `pip install -e ./python`

Una vez instalada el módulo anterior, la siguiente celda os mostrará los vuelos registrados sobre la península ibérica en estos momentos. Observad con detenimiento las propiedades del diccionario de cada vuelo.

De la misma manera que Twitter, la API de OpenSky también tiene algunas restricciones a su uso, podéis consultar-las aquí (https://openskynetwork.github.io/opensky-api/rest.html#limitations). Estas se limitan a 100 peticiones por día para usuarios sin registro. En caso de que no sean suficientes para completar vuestros ejercicios, debéis registraros en el sistema, para que se os facilite un usuario y password. El registro es mucho más sencillo que en Twitter.

In [1]:

```
import json
from random import sample

from opensky_api import OpenSkyApi
api = OpenSkyApi()
#alternativamente para evitar limitaciones de rate
#api = OpenSkyApi('USUARIO', 'PASSWORD')
states = api.get_states(bbox=(36.173357, 44.024422,-10.137019, 1.736138))
#recuperamos codigo, pais_origen, long, lat, altitud, velocidad, ratio_vertical
#atención en este ejemplo solo estamos mostrando 5 vuelos aleatorios,
#en vuestros ejercicios deberéis eliminar la función sample
for s in sample(states.states,5):
    vuelo_dict = {
                'callsign':s.callsign,
                'country': s.origin_country,
                'longitude': s.longitude,
                'latitude': s.latitude,
                'velocity': s.velocity,
                'vertical_rate': s.vertical_rate,
            }
    vuelo_encode_data = json.dumps(vuelo_dict, indent=2).encode('utf-8')
    print("(%r, %r,%r, %r, %r, %r)" % (s.callsign, s.origin_country, s.longitude, s.latitude,s.velocity,s.vertical_rate))
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-1-8769c3c56232> in <module>
     10 #atención en este ejemplo solo estamos mostrando 5 vuelos aleatorios,
     11 #en vuestros ejercicios deberéis eliminar la función sample
---> 12 for s in sample(states.states,5):
     13     vuelo_dict = {
     14                 'callsign':s.callsign,

AttributeError: 'NoneType' object has no attribute 'states'
```

Salida de ejemplo:

```
 ('BAW457  ', 'United Kingdom',-3.5196, 40.4292, 86.45, 10.73)
('BLX245  ', 'Sweden',-6.0307, 43.8266, 252.51, 0)
('CFG1HE  ', 'Germany',-8.4689, 40.2967, 236.56, 0)
('TOM3MK  ', 'United Kingdom',-7.2687, 41.5878, 247.02, 0)
('AEA57MC ', 'Spain',-0.5364, 38.2791, 64.7, -3.9)
```

Ahora vamos a crear un programa en Python para poder enviar cada 10 segundos a nuestro broker de Kafka un mensaje para cada vuelo sobre la península ibérica.

NOTA: debéis usar el mismo tópico usando antes, asegurándose de que no contiene mensajes anteriores. Para que no se líen los datos entre ejercicios podéis borrar y crear el tópico otra vez.

> **Pregunta 1. (1 punto)** Modifica el programa Python para enviar mensajes a Kafka de los datos de los vuelos en formato JSON. Os podéis auxiliar de la función json.dumps (https://docs.python.org/3/library/json.html) que nos permite crear un JSON binario de cada diccionario con las propiedades del vuelo.

Recordad como habéis creado el producer en los ejercicios de la parte 1.

Esquema

```
from time import sleep
import socket
import json
from opensky_api import OpenSkyApi
from kafka import KafkaProducer

api = OpenSkyApi()
#alternativamente para evitar limitaciones de rate
#api = OpenSkyApi('USUARIO', 'PASSWORD')
producer=KafkaProducer(bootstrap_servers=<FILL IN>)

while True:

    while(True):
        v = {}
        states = api.get_states(bbox=(36.173357, 44.024422,-10.137019, 1.736138))
        if states is not None:
            for vuelo in states.states:
                <FILL IN>
                print(v)
                producer.send(<FILL IN>,key=b'UOC', value=bytes(str(v),'utf-8'))
        sleep(10)
        producer.flush()
```

In [ ]:

```
from time import sleep
import socket
import json
from opensky_api import OpenSkyApi
from kafka import KafkaProducer

#api = OpenSkyApi()
#alternativamente para evitar limitaciones de rate
api = OpenSkyApi('vripollr', '268zr8gs3s')
producer=KafkaProducer(bootstrap_servers='Cloudera02:9092')

while True:

    while(True):
        v = {}
        states = api.get_states(bbox=(36.173357, 44.024422,-10.137019, 1.736138))
        if states is not None:
            for vuelo in states.states:
                v = {
                'callsign':vuelo.callsign,
                'country': vuelo.origin_country,
                'longitude': vuelo.longitude,
                'latitude': vuelo.latitude,
                'velocity': vuelo.velocity,
                'vertical_rate': vuelo.vertical_rate,
                    }
                #v = json.dumps(v).encode('utf-8')
                print(v)
                producer.send('PEC5vripollr',key=b'UOC', value=bytes(str(v),'utf-8'))
        sleep(10)
        producer.flush()
```

```
{'latitude': 39.484, 'vertical_rate': -0.33, 'longitude': -6.8487, 'callsign': 'TAP841Z ', 'country': 'Portugal',
'velocity': 205.45}
{'latitude': 38.8005, 'vertical_rate': -3.58, 'longitude': -9.1257, 'callsign': 'TAP282  ', 'country': 'Portuga
l', 'velocity': 71.94}
{'latitude': 36.9134, 'vertical_rate': 0, 'longitude': -3.6261, 'callsign': 'ANE26LP ', 'country': 'Spain', 'velo
city': 121.06}
{'latitude': 40.1906, 'vertical_rate': 0, 'longitude': -7.9892, 'callsign': 'VOE74MR ', 'country': 'Spain', 'velo
city': 179.22}
{'latitude': 39.2093, 'vertical_rate': 0, 'longitude': -4.374, 'callsign': 'EJU96HM ', 'country': 'Austria', 'vel
ocity': 217.04}
{'latitude': 39.3276, 'vertical_rate': -5.53, 'longitude': -8.7354, 'callsign': 'TAP1061 ', 'country': 'Estonia',
'velocity': 146.11}
{'latitude': 38.8981, 'vertical_rate': -2.93, 'longitude': 1.4343, 'callsign': 'IBS3812 ', 'country': 'Spain', 'v
elocity': 62.21}
{'latitude': 40.0434, 'vertical_rate': 14.31, 'longitude': -4.2145, 'callsign': 'AEA11YP ', 'country': 'Spain',
'velocity': 190.8}
{'latitude': 40.4719, 'vertical_rate': 14.31, 'longitude': -3.3136, 'callsign': 'WZZ3273 ', 'country': 'Hungary',
'velocity': 134.52}
{'latitude': 39.5948, 'vertical_rate': 0, 'longitude': 1.2332, 'callsign': 'AEA56MF ', 'country': 'Spain', 'veloc
```

> **Pregunta 2. (1 punto)** Se pide leer los mensajes almacenados en Kafka mediante structured streaming y mostrar el esquema de los datos recibidos. En este primer ejercicio se pide mostrar el esquema usando printSchema() y mostrar los datos recibidos.

NOTA Es importante que antes de mostrar los datos hagáis una conversión de estos a string, ya que llegan en formato binario. Esta conversión se puede realizar de forma fácil con la siguiente expresión.

```
vuelosDF.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")\
```

> Una vez comprobada que la transmisión funciona, se pide realizar un pre-procesado antes del envío de los datos a Kafka para eliminar aquellas líneas de datos que no sean útiles ni convenientes.

NOTA para la configuración de Kafka como input sink en Structured Streaming podéis revisar la documentación de los sinks (https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#input-sources) y la guía de intergración (https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html).

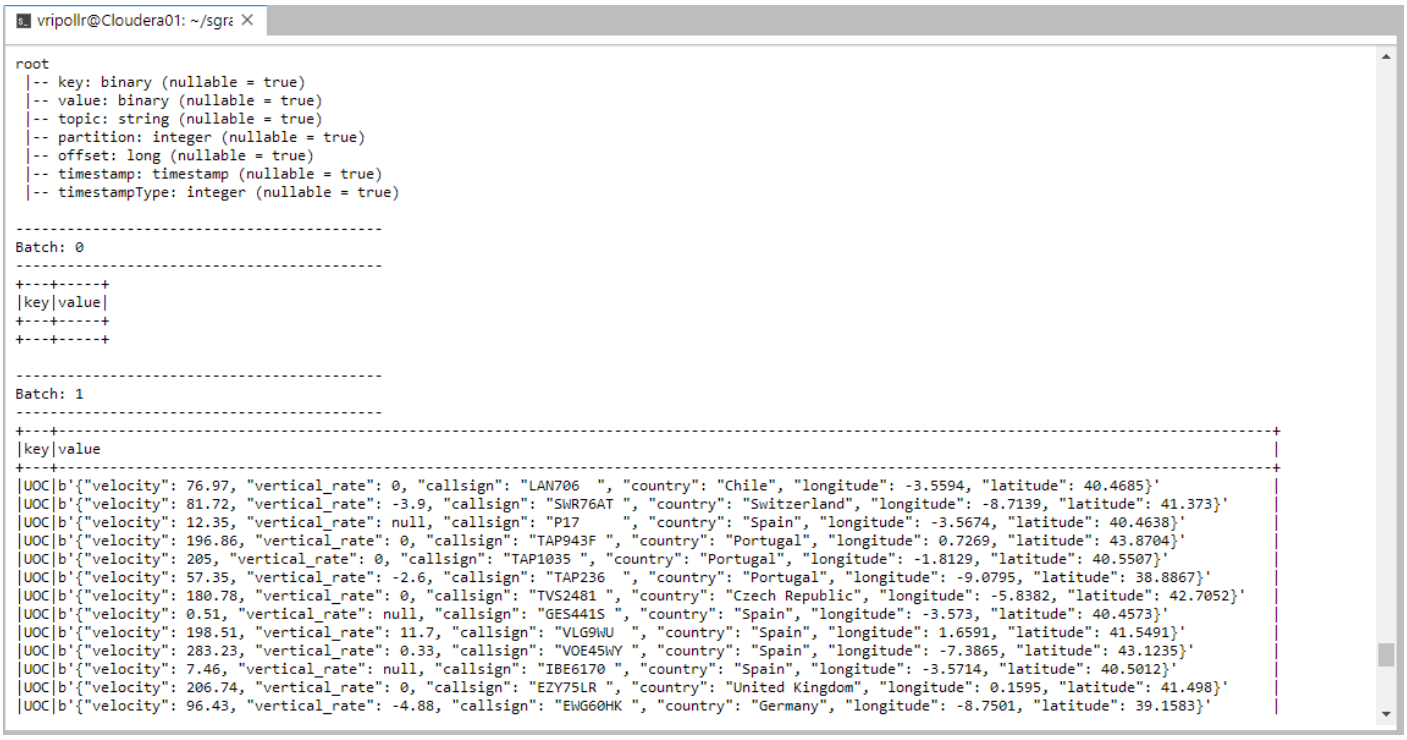
Salida de ejemplo:

```
root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)

True
-------------------------------------------
Batch: 0
-------------------------------------------
+---+-----+
|key|value|
+---+-----+
+---+-----+

-------------------------------------------
Batch: 1
-------------------------------------------
+---+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------+
|key|value                                                                                                                                                                  |
+---+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------+
|UOC|b'{"latitude": 40.1195, "vertical_rate": -9.75, "callsign": "TAP553R ", "longitude": -7.9077, "velocity": 209.9, "country": "Portugal"}'|
|UOC|b'{"latitude": 42.1781, "vertical_rate": 0, "callsign": "TAP58Y  ", "longitude": -8.3566, "velocity": 227.67, "country": "Portugal"}'    |
|UOC|b'{"latitude": 41.7621, "vertical_rate": 0, "callsign": "TAP572F ", "longitude": -5.858, "velocity": 215.53, "country": "Portugal"}'     |
|UOC|b'{"latitude": 41.2381, "vertical_rate": -7.48, "callsign": "TAP1046 ", "longitude": 0.9924, "velocity": 199.26, "country": "Portugal"}'|
|UOC|b'{"latitude": 42.9246, "vertical_rate": -0.33, "callsign": "VOE46NT ", "longitude": -7.603, "velocity": 211.42, "country": "Spain"}'    |
|UOC|b'{"latitude": 39.1773, "vertical_rate": -16.26, "callsign": "IBS3812 ", "longitude": -0.0438, "velocity": 202.28, "country": "Spain"}' |
|UOC|b'{"latitude": 41.2983, "vertical_rate": -19.51, "callsign": "IBE30VR ", "longitude": -0.0531, "velocity": 216.81, "country": "Spain"}' |
|UOC|b'{"latitude": 38.9916, "vertical_rate": -1.63, "callsign": "IBE31LF ", "longitude": -8.9668, "velocity": 110.06, "country": "Spain"}'  |
|UOC|b'{"latitude": 43.8304, "vertical_rate": 0, "callsign": "WZZ1270 ", "longitude": 0.6341, "velocity": 236.94, "country": "Hungary"}'     |
```

In [ ]:

```
import time
import sys
import findspark
findspark.init()

from pyspark import SparkConf, SparkContext, SQLContext, HiveContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

conf = SparkConf()
conf.setMaster("local[2]") #En local con dos threads
sc = SparkContext(conf=conf)

#Introducid el nombre de la app PEC5_ seguido de vuestro nombre de usuario

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Uso: PEC5_2_1_1 localhost 20216", file=sys.stderr)
        exit(-1)

    host = sys.argv[1]
    port = int(sys.argv[2])

    spark = SparkSession\
        .builder\
        .appName('PEC5_vripollr')\
        .getOrCreate()

    #Creamos el dataframe representando el stream de linias del netcat desde host:port
    vuelosdf = spark.readStream.format("kafka").option("kafka.bootstrap.servers", "Cloudera02:9092").option("subscribe", "PEC5vri

    #Mostramos esquema
    vuelosdf.printSchema()

    #Ejecutamos consulta
    vuelosdf.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)") \
    .writeStream \
    .option("truncate", "false")\
    .format("console") \
    .outputMode("append") \
    .start() \
    .awaitTermination()
```

Copia la salida obtenida en formato de imagen:



Pregunta 3. (1 punto) Se pide mostrar la información en forma de tabla, con las columnas, country|callsign|longitude|latitude|velocity|vertical_rate. Para ello vais a tener que crear un esquema mediante StructType (https://spark.apache.org/docs/3.2.1/api/python/reference/api/pyspark.sql.types.StructType.html) y aplicarlo a la función SQL from_json (https://spark.apache.org/docs/3.2.1/api/python/reference/api/pyspark.sql.functions.from_json.html). De esta manera podremos pasar de una columna string con todo el JSON, a 6 columnas con el tipo ajustado al valor contenido.

Salida de ejemplo:

```
-------------------------------------------
Batch: 1
-------------------------------------------
+--------------------+--------+---------+--------+--------+-------------+
|             country|callsign|longitude|latitude|velocity|vertical_rate|
+--------------------+--------+---------+--------+--------+-------------+
|            Portugal|TAP11WQ |  -3.6832|  40.639|  124.26|        15.61|
|            Portugal|TAP1040 |  -6.7756| 39.6546|  242.33|         0.0|
|      United Kingdom|EZY52AG |  -6.9604| 42.2392|  212.11|        -0.33|
|            Portugal|TAP016  |  -9.0759| 38.8932|   77.48|        -4.23|
|              Greece|SXAQG   |  -3.3046| 37.6474|  202.04|       -13.33|
|               Spain|IBE3195 |   -1.798| 40.5692|  228.67|       -14.96|
|             Hungary|WZZ1708 |  -6.6787| 37.3123|  217.19|        -0.33|
|      United Kingdom|EZY82AW |  -0.1846| 40.2124|  181.16|        -8.78|
|      Czech Republic|TVS2480 |  -5.9276| 41.1227|  224.84|         0.0|
|      United Kingdom|EXS34UV |  -7.9631| 37.0183|    2.06|        null|
|      United Kingdom|EXS69G  |  -7.5466| 41.4589|  214.85|         0.0|
|               Spain|VLG6CT  |  -2.6752| 37.3665|  220.14|         5.2|
|Kingdom of the Ne...|KLM61Z  |  -2.7241|  40.902|  172.92|         -6.5|
|             Ireland|RYR1LZ  |  -7.9804| 37.0165|   41.16|        null|
|      United Kingdom|EZY29CE |  -9.1985| 41.2836|  207.54|         0.0|
|      United Kingdom|EZY35UF |  -7.2926| 41.5363|   221.0|         0.33|
```

In [ ]:

```
import json
import time
import sys
import findspark
findspark.init()

from pyspark.sql.types import StructType, StructField, StringType, IntegerType, TimestampType, LongType, DoubleType
from pyspark import SparkConf, SparkContext, SQLContext, HiveContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

conf = SparkConf()
conf.setMaster("local[2]") #En local con dos threads
sc = SparkContext(conf=conf)

#Introducid el nombre de la app PEC5_ seguido de vuestro nombre de usuario

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Uso: PEC5_2_1_1 localhost 20216", file=sys.stderr)
        exit(-1)

    host = sys.argv[1]
    port = int(sys.argv[2])

    spark = SparkSession\
        .builder\
        .appName('PEC5_vripollr')\
        .getOrCreate()

    schema = StructType([
    StructField("longitude",DoubleType(),True),
    StructField("vertical_rate",StringType(),True),
    StructField("country",StringType(),True),
    StructField("callsign",StringType(),True),
    StructField("velocity",DoubleType(),True),
    StructField("latitude",DoubleType(),True)])

    vuelosdf1 = spark.readStream.format("kafka").option("kafka.bootstrap.servers", "Cloudera02:9092").option("subscribe", "PEC5vr
    vuelosdf2 = vuelosdf1.selectExpr("CAST(value AS STRING)")
    vuelosdf3 = vuelosdf2.select(from_json(col("value"), schema).alias("data")).select("data.*")

    vuelosdf3.writeStream\
      .format("console")\
      .outputMode("append")\
      .option('truncate', 'false')\
      .start()\
      .awaitTermination()
```

Copia la salida obtenida en formato imagen:

vripollr@Cloudera01: ~/sgra

```
-------------------------------------------
+---------+-------------+-------+--------+--------+--------+
|longitude|vertical_rate|country|callsign|velocity|latitude|
+---------+-------------+-------+--------+--------+--------+
+---------+-------------+-------+--------+--------+--------+

-------------------------------------------
Batch: 1
-------------------------------------------
+---------+-------------+--------+--------+--------+--------+
|longitude|vertical_rate|country |callsign|velocity|latitude|
+---------+-------------+--------+--------+--------+--------+
|-8.1813  |-10.4        |Germany |SCR168  |197.49  |37.9707 |
|-9.1454  |-3.58        |Portugal|TAP789  |66.88   |38.7636 |
|-1.9159  |0            |Portugal|TAP431  |204.75  |43.264  |
|-5.0476  |0            |Austria |EJU63ME |245.27  |41.5617 |
|-9.844   |4.55         |Portugal|TAP083  |197.79  |37.8612 |
|-3.742   |0            |Portugal|TAP282  |223.72  |36.6084 |
|-9.202   |13.66        |Portugal|TAP243U |137.39  |38.9677 |
|1.4257   |-2.93        |Spain   |ANE29RH |52.93   |38.8945 |
|-2.8436  |10.73        |Hungary |WZZ533  |214.48  |40.9332 |
|-6.6088  |0            |Estonia |TAP1061 |125.24  |41.2152 |
|null     |null         |null    |null    |null    |null    |
|null     |null         |null    |null    |null    |null    |
|-3.3262  |-12.35       |Hungary |WZZ1613 |201.63  |37.3712 |
|-4.9231  |0            |Spain   |IBE3342 |253.85  |36.7306 |
|-8.7404  |-4.23        |Spain   |SWT960P |135.78  |38.664  |
|null     |null         |null    |null    |null    |null    |
|-0.7602  |5.85         |Spain   |VLG8257 |134.92  |38.3224 |
|-0.2148  |0.33         |Spain   |VLG1863 |231.47  |39.6811 |
|null     |null         |null    |null    |null    |null    |
|1.7259   |8.45         |Spain   |VLG4QD  |202.67  |41.8383 |
+---------+-------------+--------+--------+--------+--------+
only showing top 20 rows
```

Pregunta 4. (1 punto) Muestra el total de vuelos para cada destino agrupados por país de destino que hay en cada momento. Los datos deben mostrarse ordenados por país alfabéticamente. Tened en cuenta que podemos recibir datos duplicados dado que el script de OpenSky lee cada 10 segundos todos los vuelos existentes y los envía a Kafka. Por defecto Spark crea 200 tareas (cada una implicará una partición de los datos) por stage en el procesamiento en Structured Streaming. Para acelerar el proceso de captura, se pide que ajustéis (https://spark.apache.org/docs/latest/sql-performance-tuning.html#other-configuration-options) el parámetro en la configuración de SparkSession a un valor de 4 particiones.

Salida de ejemplo:

```
+--------------------+-----+
|             country|count|
+--------------------+-----+
|             Austria|   10|
|             Bahrain|    1|
|             Belgium|    6|
|            Bulgaria|    1|
|      Czech Republic|    2|
|             Denmark|    2|
|             Estonia|    1|
|              France|    8|
|             Germany|   21|
|              Greece|    1|
|             Hungary|    1|
|             Ireland|   22|
|              Jordan|    1|
|Kingdom of the Ne...|    4|
|              Kuwait|    1|
|              Latvia|    1|
|               Malta|   13|
|             Morocco|   18|
|              Poland|    2|
|            Portugal|   15|
+--------------------+-----+
```

In [ ]:

```
import json
import time
import sys
import findspark
findspark.init()

from pyspark.sql.types import StructType, StructField, StringType, IntegerType, TimestampType, LongType, DoubleType
from pyspark import SparkConf, SparkContext, SQLContext, HiveContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

conf = SparkConf()
conf.setMaster("local[2]") #En local con dos threads
sc = SparkContext(conf=conf)

#Introducid el nombre de la app PEC5_ seguido de vuestro nombre de usuario

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Uso: PEC5_2_1_1 localhost 20216", file=sys.stderr)
        exit(-1)

    host = sys.argv[1]
    port = int(sys.argv[2])

    spark = SparkSession\
        .builder\
        .appName('PEC5_vripollr')\
        .getOrCreate()

    #Ajustamos en número de particiones a 4
    spark.conf.set("spark.sql.shuffle.partitions",4)

    schema = StructType([
    StructField("longitude",DoubleType(),True),
    StructField("vertical_rate",StringType(),True),
    StructField("country",StringType(),True),
    StructField("callsign",StringType(),True),
    StructField("velocity",DoubleType(),True),
    StructField("latitude",DoubleType(),True)])

    vuelosdf1 = spark.readStream.format("kafka").option("kafka.bootstrap.servers", "Cloudera02:9092").option("subscribe", "PEC5vr
    vuelosdf2 = vuelosdf1.selectExpr("CAST(value AS STRING)")
    #vuelosdf2.createOrReplaceTempView("Tabla")
    #vuelosdf3 = spark.sql("select value from Tabla")
    #vuelosdf3.printSchema()
    vuelosdf3 = vuelosdf2.select(from_json(col("value"), schema).alias("data")).select("data.*")
    vuelosdf4 = vuelosdf3.groupBy(col('country')).count().orderBy('country').filter(col('country')!='null')

    vuelosdf4.writeStream\
      .format("console")\
      .outputMode("complete")\
      .option('truncate', 'false')\
      .start()\
      .awaitTermination()
```

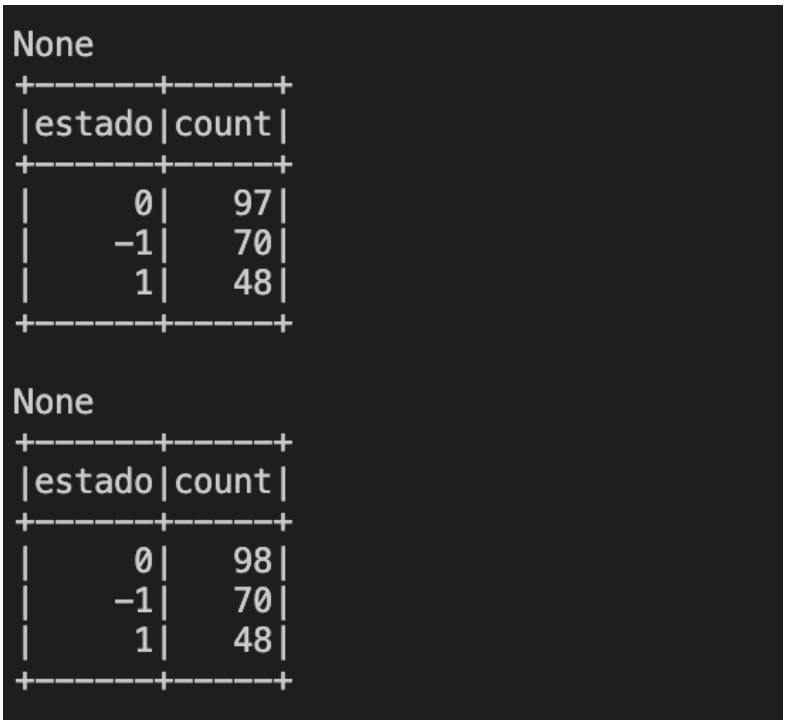
Copia la salida obtenida en formato de imagen:

vripollr@Cloudera01: ~/sgra

```
|Poland                    |3    |
|Portugal                  |64   |
|Spain                     |105  |
+--------------------------+-----+
only showing top 20 rows

-------------------------------------------
Batch: 7
-------------------------------------------
+--------------------------+-----+
|country                   |count|
+--------------------------+-----+
|Algeria                   |4    |
|Angola                    |4    |
|Argentina                 |4    |
|Austria                   |56   |
|Belgium                   |8    |
|Czech Republic            |4    |
|Denmark                   |4    |
|Estonia                   |4    |
|France                    |28   |
|Germany                   |48   |
|Hungary                   |16   |
|Ireland                   |124  |
|Kingdom of the Netherlands|17   |
|Luxembourg                |4    |
|Malta                     |36   |
|Morocco                   |40   |
|Norway                    |28   |
|Poland                    |4    |
|Portugal                  |80   |
|Spain                     |140  |
+--------------------------+-----+
only showing top 20 rows
```

Pregunta 5. (1 punto) Agrupa todos los vuelos que están subiendo en altura, los que están bajando y los que están en tierra. Indica su número. Deberás auxiliarte de una consulta SQL para poder indicar con -1 que un vuelo está descendiendo, +1 si está subiendo, y 0 si está en tierra.

Salida de ejemplo:

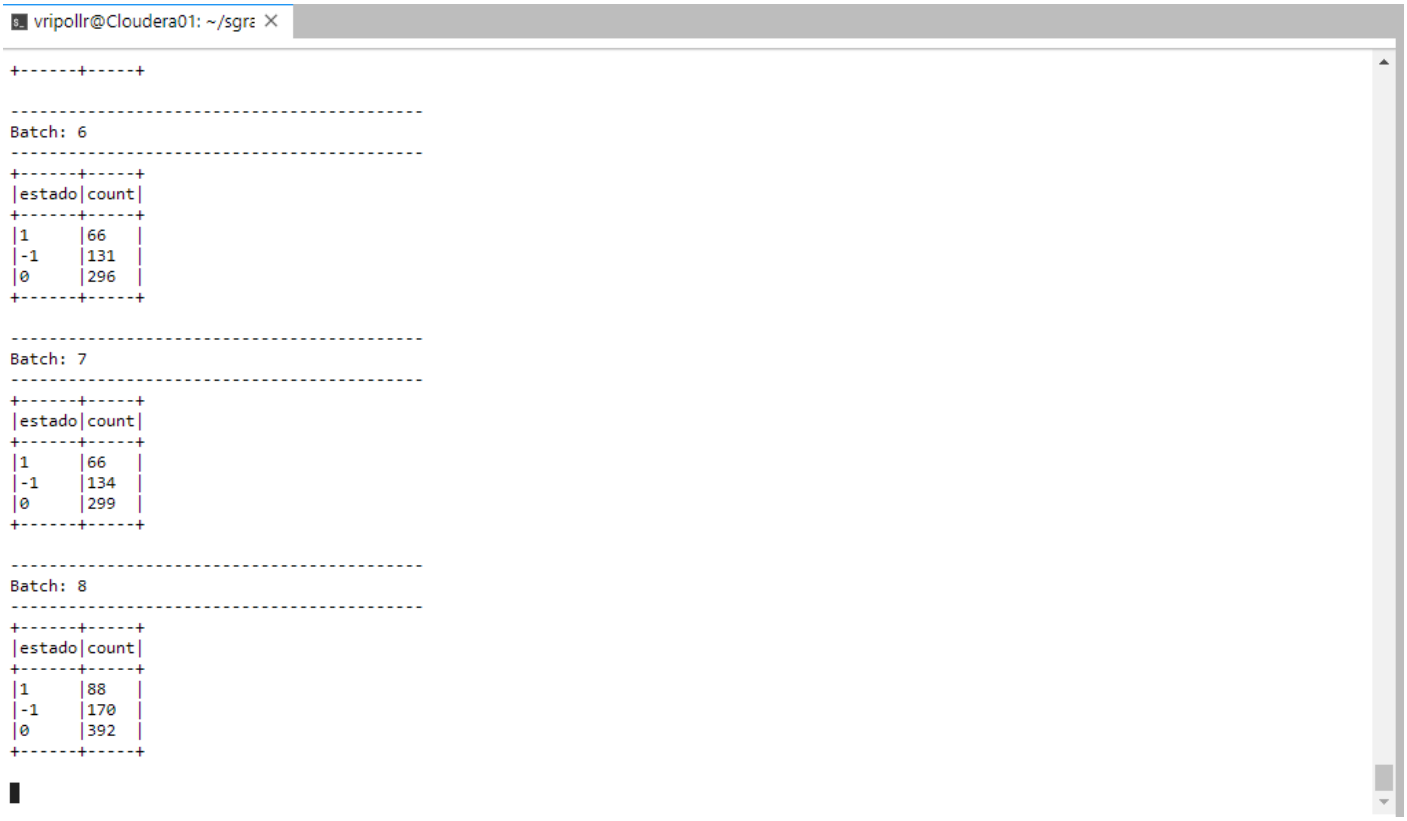


```
None
+------+-----+
|estado|count|
+------+-----+
|     0|   97|
|    -1|   70|
|     1|   48|
+------+-----+

None
+------+-----+
|estado|count|
+------+-----+
|     0|   98|
|    -1|   70|
|     1|   48|
+------+-----+
```

In [ ]:

```
import json
import time
import sys
import findspark
findspark.init()

from pyspark.sql.types import StructType, StructField, StringType, IntegerType, TimestampType, LongType, DoubleType
from pyspark import SparkConf, SparkContext, SQLContext, HiveContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

conf = SparkConf()
conf.setMaster("local[2]") #En local con dos threads
sc = SparkContext(conf=conf)

#Introducid el nombre de la app PEC5_ seguido de vuestro nombre de usuario

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Uso: PEC5_2_1_1 localhost 20216", file=sys.stderr)
        exit(-1)

    host = sys.argv[1]
    port = int(sys.argv[2])

    spark = SparkSession\
        .builder\
        .appName('PEC5_vripollr')\
        .getOrCreate()

    #Ajustamos en número de particiones a 4
    spark.conf.set("spark.sql.shuffle.partitions",4)

    schema = StructType([
    StructField("longitude",DoubleType(),True),
    StructField("vertical_rate",StringType(),True),
    StructField("country",StringType(),True),
    StructField("callsign",StringType(),True),
    StructField("velocity",DoubleType(),True),
    StructField("latitude",DoubleType(),True)])

    vuelosdf1 = spark.readStream.format("kafka").option("kafka.bootstrap.servers", "Cloudera02:9092").option("subscribe", "PEC5vr
    vuelosdf2 = vuelosdf1.selectExpr("CAST(value AS STRING)")
    vuelosdf3 = vuelosdf2.select(from_json(col("value"), schema).alias("data")).select("data.*")
    vuelosdf4 = vuelosdf3.select(col('vertical_rate'))\
                .withColumn('vertical_rate', when(col('vertical_rate')<0,'-1')\
                .when(col('vertical_rate')==0,'0').otherwise(when(col('vertical_rate')>0,'1')))\
                .withColumnRenamed('vertical_rate','estado')\
                .filter(col('estado')!='null')\
                .groupBy('estado')\
                .count()
    vuelosdf4.writeStream\
      .format("console")\
      .outputMode("complete")\
      .option('truncate', 'false')\
      .start()\
      .awaitTermination()
```

Copia la salida obtenida en formato de imagen:



Pregunta 6. (1 punto) ¿De qué manera podemos identificar que aparece un nuevo vuelo en el espacio aéreo?. No hace falta escribir el código sino describir con palabras como se plantearía la solución.

In [ ]:

```
#En primer lugar habría que recurrir a alguna api como la que se ha estudiado en esta práctica, pero existen más como ADS-B
#exchange o Apilayer Aviationstack. El siguiente paso sería crear un topic de Kafka al cual poder enviar los datos que nos
#resultasen útiles. A continuación con python podríamos recibir los eventos para procesarlos. Para el propósito, identificar
#nuevos vuelos, una opción sería mostrar los datos en forma de tabla tal y como se ha hecho en los ejercicios anteriores.
#Como nos interesa detectar solo nuevos vuelos una opción muy básica sería recurrir al output mode complete para mostrar
#todos los datos recibidos en orden y fijarnos en las últimas filas. Otra opción sería utilizar el output mode append,
#definiendo previamente un trigger que nos sirviese como punto de partida para contar nuevos vuelos.
#Otra opción podría ser buscar en la información de los vuelos algún dato relacionado con la hora en la que ha salido,
#si lo hubiese, y ordenar los vuelos en función de este parámetro.
```

Pregunta 7. (1 punto) Explica brevemente en una extensión de entre 5 y 10 líneas las ventajas e inconvenientes de utilizar Structured Streaming versus Spark Streaming.

In [ ]:

```
#Spark streaming se basa en los clásicos RDD por lo que la forma de trabajar en este entorno es muy parecida a la habitual con
#Spark. Utiliza la API DStream. Structured streaming se basa en trabajar sobre dataframes y datasets de la librería Spark SQL,
#por lo que es más útil si se quieren aplicar consultas sql. Por lo general trabajar con dataframes resulta más sencillo y
#se dispone de más operaciones [1]. Un problema importante del procesado de datos en stream es la gestión del tiempo asociado
#a cada evento y en especial, gestionar aquellos eventos que llegan con retardo. Con structured streaming se puede trabajar con
#event-time windows. Este modelo permite agregar eventos en los dataframes de forma que aunque un evento llegue con retardo
#este sea ordenado en base a la hora de su creación, no la de llegada. En spark streaming esto no es posible [2]. Ambos modelos
#son tolerantes a fallos [3][4], sin embargo la división de los datos en RDD es más lenta que el procesado de Dataframes lo que
#incrementa aún más el problema de la latencia de Spark streaming.
#
#[1] https://blog.knoldus.com/spark-rdd-vs-dataframes/
#[2] Apuntes: Procesado de datos en streaming con Spark Streaming, Structured Streaming y Storm
#[3] https://spark.apache.org/docs/2.4.3/structured-streaming-programming-guide.html
#[4] https://spark.apache.org/docs/latest/streaming-programming-guide.html
```