Procesado de datos en *streaming* con Spark Streaming, Structured Streaming y Storm

PID_00276699

Alberto Álvarez Vales

Tiempo mínimo de dedicación recomendado: 3 horas

Universitat Oberta de Catalunya

Alberto Álvarez Vales

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Albert Solé

Primera edición: marzo 2021

Av. Tibidabo, 39-43, 08035 Barcelona
Autoría: Alberto Álvarez Vales
Producción: FUOC

Índice

1. *Stream processing* 5
1.1. Las tecnologías de implementación 5

2. El procesado de *streaming* con Spark 6

3. Spark Streaming 7
3.1. DStreams 7
3.2. Los DStreams de entrada 8
3.3. Las transformaciones sobre DStreams 8
3.4. Las operaciones de salida en DStreams 10
3.5. Las operaciones *window* 10
3.6. Las operaciones *join* 12
3.7. La persistencia 12
3.8. *Checkpointing* 13
3.9. *Fault-tolerance semantics* 14
3.10. Ejemplo básico 15

4. Spark Structured Streaming 19
4.1. Los conceptos básicos 19
4.2. Ejemplo de Spark Structured Streaming 21
4.3. Las fuentes de entrada 23
4.4. Las salidas de *streaming* 24
4.5. Las transformaciones 25
4.6. Las ventanas con *event time* y *watermarking* 26
4.7. Las operaciones *join* 28
4.8. El caso de uso 29

5. Apache Storm 32
5.1. La arquitectura: los componentes de Storm 32
5.2. El modelo de datos de Storm 33
5.3. La implementación 34

Bibliografía 39

1. *Stream processing*

1.1. Las tecnologías de implementación

Como ya hemos visto en los módulos teóricos, definimos *stream processing* como el conjunto de técnicas que se utilizan para capturar, procesar y almacenar información de conjuntos de datos ilimitados (*unbounded data*) llamados *datos en flujo*. Existen multitud de herramientas de *stream processing*, pero en nuestro caso centraremos el análisis en Spark (en sus dos versiones) y en Storm.

La razón principal de escoger estas dos herramientas se fundamenta en sus características técnicas y de arquitectura, ya que ofrecen aproximaciones al problema muy distintas. Spark es una tecnología que permite tanto el procesado en *batch*, como el procesado en *streaming*, pero mantiene las API de *batch* y *streaming* de forma separada. Storm está especializada en el procesado en *streaming* y es la tecnología básica que se usa en la arquitectura Lambda (parte de la *speed layer*).

Primero de todo, se debe indicar que Spark y Storm tienen tres características comunes:

- Los datos que se reciben en *streaming* serán la entrada (*input source*) para los algoritmos de procesado que se apliquen.
- La aplicación se envía a un nodo en el clúster que la ejecuta.
- Los algoritmos de procesado que implementemos se ejecutan de forma distribuida entre todos los nodos que forman el clúster. Esto se hace de forma transparente al usuario.

En los siguientes apartados vamos a entrar en detalle con cada una de estas tecnologías. Se verán las diferencias entre cada una de ellas y, finalmente, para cada uno de los conceptos descritos vamos a implementar (y a explicar) ejemplos concretos.

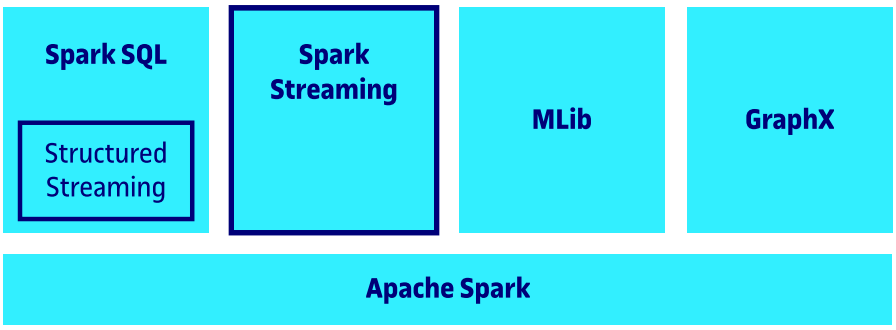
2. El procesado de *streaming* con Spark

Spark ofrece dos mecanismos distintos para el procesado de datos en *streaming*. Uno de ellos está basado en los clásicos *resilient distributed dataset* (RDD), que de alguna manera ofrecen una interfaz muy parecida a la habitual de Spark. La segunda corresponde a una aproximación técnica mucho más actual y en la dirección de poder prescindir de la capa *batch*, ofreciendo propiedades de *exactly once* en el procesado.

- **Spark Streaming.** Se sustenta en los RDD de Spark. Utiliza para ello una librería específica que proporciona la API DStream.

- **Structured Streaming**. Basada en el uso de los *dataframes* y *datasets* de la librería Spark SQL. En este caso ofrece la posibilidad de ejecutar sentencias SQL en *streaming*.

Figura 1. Librerías de *stream processing* en Spark: Spark Streaming y Structured Streaming como parte de Spark SQL



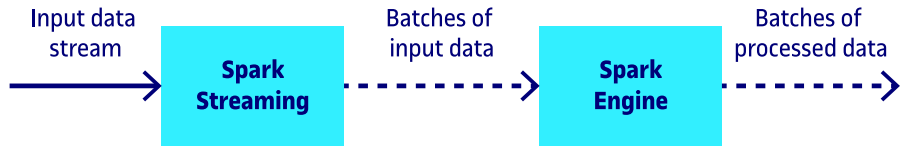
Fuente: elaboración propia.

3. Spark Streaming

Spark Streaming es una parte de las librerías nativas de Spark para procesar, de forma continuada, flujos de datos en *streaming*. Se ofrece desde la versión 0.7 que salió en el año 2013. Por tanto, podemos decir que es una tecnología bastante consolidada dentro del entorno de procesado en *streaming*.

La API proporciona la estructura básica de DStream que se sustenta sobre los conocidos RDD de Spark, usados en el procesado *batch*. DStreams proporciona los datos divididos en porciones, similar a los RDD recibidos desde la fuente de *streaming*. Este tipo de funcionamiento suele llamarse *micro-batching*.

Figura 2. Flujo de procesado de Spark Streaming



Fuente: http://spark.apache.org/docs/latest/streaming-programming-guide.html

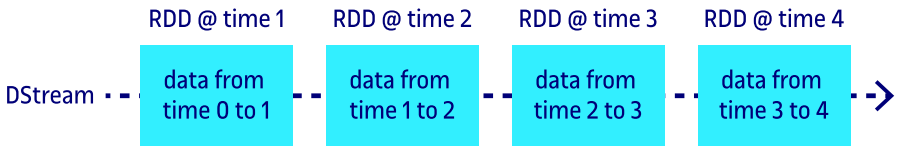
A partir del flujo de datos de entrada, Spark Streaming preprocesa los datos para dividirlos en los citados *micro-batches* ('bloques de datos pequeños', en castellano) según el tiempo predefinido (este punto, como veremos, es configurable). A continuación, el motor Spark (*spark engine*) se encarga de aplicar las operaciones predefinidas a cada uno de los bloques de datos (de forma muy similar al procesado *batch*), proporcionando un resultado para cada uno de ellos que se retorna al usuario (figura 2). En resumen, lo que hace Spark Streaming a partir de un flujo de datos continuo es convertirlo en un flujo discreto, llamado DStream, para que el *spark engine* lo pueda procesar.

Spark Streaming puede consumir datos de multitud de fuentes, como Kafka, Flume, HDFS, ZeroMQ o Twitter, además de poder personalizar la API para poder trabajar con otras fuentes de datos.

3.1. DStreams

Internamente un DStream se representa como una serie continuada de RDD. Cada RDD contiene datos de un intervalo de tiempo, tal y como se aprecia en la siguiente figura:

Figura 3. Segmentación del flujo de datos en *micro-batches*



Fuente: http://spark.apache.org/docs/latest/streaming-programming-guide.html

Como vimos en el módulo «Procesado de datos en *streaming*» y como veremos en los siguientes apartados, el formato de las ventanas es configurable. Aun así, Spark Streaming solo permite ventanas de tiempo y no de eventos.

A continuación, vamos a ver cómo Spark Streaming soluciona las tres etapas del procesado de datos en *streaming*, que son las siguientes: captura y preprocesado y procesado y almacenamiento de los resultados.

3.2. Los DStreams de entrada

Los DStreams son la estructura básica en que se basa todo el procesado mediante Spark Streaming. De la misma forma que en los RDD hay varias formas de construirlos, mediante transformaciones y acciones (como veremos a continuación) o mediante fuentes externas de datos. Mediante este último método, Spark Streaming permite capturar datos de varias fuentes primarias. Las podemos clasificar en las tres categorías siguientes:

- **Fuentes básicas**. Están disponibles directamente en la API del *streaming context*. Son conexiones mediante *sockets* y sistemas de archivos.

- **Fuentes avanzadas.** Requieren conectar Spark a herramientas de captura específicas mediante dependencias externas. En este punto podemos citar a Kafka y a Kinesis.

- **Fuentes configurables.** Para otras fuentes de *streaming* se puede programar un receptor, pero solo puede hacerse en los lenguajes Java o Scala.

3.3. Las transformaciones sobre DStreams

Las transformaciones y acciones ofrecidas por los DStreams son muy similares a las existentes con RDD. Los DStreams soportan muchas de las transformaciones que se pueden hacer con RDD y algunas específicas en relación con la tipología de datos que procesar. A continuación, se presenta un listado de las principales transformaciones:

- **`.map`**`(func)`. Devuelve un nuevo DStream en el que se aplica la función `func` a cada elemento del DStream original.

- **`.flatMap`**(*`func`*). Es similar a *map*, pero flatMap extiende el resultado haciendo que, por ejemplo, cada elemento de una lista se convierta en un nuevo elemento del DStream.

- **`.filter`**(*`func`*). Crea un nuevo DStream con los elementos que satisfacen los criterios del filtro.

- **`.repartition`**(*`numParticiones`*). Aumenta o reduce el paralelismo, cambiando el número de particiones.

- **`.union`**(*`otroDStream`*). Devuelve un nuevo DStream que contiene la unión de los dos DStream.

- **`.count`**(). Devuelve el número de elementos que hay en cada uno de los RDD del DStream.

- **`.countByValue`**(). Devuelve un nuevo DStream que permite saber la frecuencia de aparición de un elemento en cada RDD del DStream de origen.

- **`.reduceByKey`**(*`func`*). Al utilizarlo en un DStream de pares (*clave*, *valor*), se devuelve un DStream en el que se aplica la función *`func`* de agregación a todos los valores que compartan la clave.

- **`.reduce`**(*`func`*). Crea un DStream en el que cada RDD tiene un elemento resultado de aplicar la función *`func`* a cada RDD del DStream de entrada.

- **`.join`**(*`otroDStream`*). Permite combinar un DStream con otro o un DStream con un RDD. Se explicará con más detalle en un apartado posterior dedicado específicamente a los diferentes tipos de `join`.

Las funciones específicas que se utilizan con DStreams son las siguientes:

1) **`updateStateByKey`**(*func*). Con esta función se pueden dar resultados acumulados, teniendo en cuenta también los resultados previos. Recordemos que la forma habitual de usar los DStreams es mediante *micro-batches* que se procesan de manera individual y no tienen relación ni compartición de datos los unos con los otros (*StateLess*).

Para que pueda funcionar es necesario haber configurado previamente una carpeta para almacenar los valores de estado, con el método *checkpoint*.

El estado para cada clave (`key`) se actualiza aplicando la función *`func`* que se pasa como parámetro. A dicha función, `updateStateByKey` se le proporciona como primer parámetro la lista de valores asociados a la misma clave y, como segundo parámetro, el estado anterior.

Para entender cómo se utiliza, se puede analizar el siguiente ejemplo:

```
#…
ssc.checkpoint("/checkpoint")
#…
def actualizaStateFul(nuevasVentas, ventasAnteriores):
 return sum(nuevasVentas) + (ventaAnteriores or 0)
#…
ventasTotales = ventas.updateStateByKey(actualizaStateFul)
```

En este ejemplo, el DStream «ventas» representa los valores de ventas procesados en el último intervalo de tiempo que se irán acumulando en el DStream `ventasTotales`.

2) **`transform`**(*func*). Esta función permite que en los DStream se puedan aplicar funciones de RDD que no estén contempladas en la DStream API. Deben ser funciones que se apliquen a un RDD y que devuelvan también un RDD, ya que se aplicarán a cada uno de los RDD que conforman el DStream.

Con el siguiente ejemplo, para cada RDD que conforma el DStream se toma tan solo la mitad de los valores, escogiéndolos de forma aleatoria mediante el método *`.sample`* que existe para los RDD, pero no para DStreams.

```
mitadDStream=miDStream.transform(lambda rdd:rdd.sample(False,0.5))
```

3.4. Las operaciones de salida en DStreams

Una vez aplicadas las operaciones necesarias a los datos de entrada, debemos llevar los resultados hacia las bases de datos o los sistemas de ficheros para almacenar los resultados. Las operaciones válidas utilizando Python son:

- **`print`**`()`. Muestra por pantalla los diez primeros elementos de datos del DStream. En general, solo se utiliza para el desarrollo y la depuración.

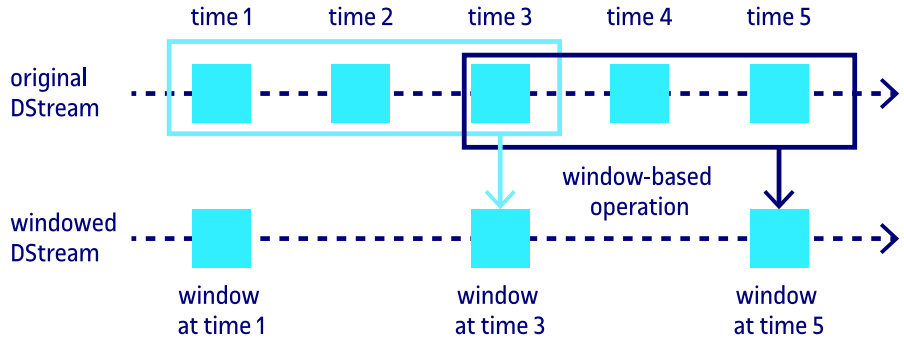
- **`saveAsTextFiles`**`(`*`prefix`*`, [`*`suffix`*`])` y **`saveAsHadoopFiles`**`(`*`prefix`*`, [`*`suffix`*`])`. Permite grabar el contenido de cada DStream que se genera en cada intervalo a un medio de almacenamiento permanente.

- **`foreachRDD`**`(`*`func`*`)`. Esta función es genérica y permite aplicar una función a cada RDD generado en el *stream*. Con esta función se podrán grabar los RDD a archivos o guardar la información en una base de datos.

3.5. Las operaciones *window*

Como hemos visto en los módulos teóricos, uno de los sistemas de procesamiento de datos en *streaming* es mediante ventanas. Spark Streaming ofrece esta posibilidad permitiendo aplicar transformaciones y acciones sobre una ventana deslizante de datos, como se puede visualizar en la figura 4:

Figura 4. Ejemplo de procesado mediante ventana deslizante en Spark Streaming



Fuente: http://spark.apache.org/docs/latest/streaming-programming-guide.html

En el caso de la figura, se muestra el procesado los datos mediante una ventana de tres unidades de tiempo y con un desplazamiento de dos unidades de tiempo. Para las operaciones con ventanas, se necesitará especificar dos parámetros: la duración de la ventana y el intervalo de tiempo que indique cada cuanto se realiza la operación.

Cuando no se utilizan ventanas, se define un intervalo de tiempo *t*, que indica cada cuanto tiempo se va a procesar el RDD correspondiente y, además, indica el intervalo de tiempo de dicho RDD; es decir, se indica la duración de cada *micro-batch*. En el ejemplo que se da en un apartado posterior, se puede comprobar que el intervalo de tiempo es de un segundo y se define en el `StreamingContext`.

```
ssc = StreamingContext(sc, 1) #intervalo de tiempo de 1 segundo
```

Si se utilizan ventanas, se pueden definir, como se ha visto en la figura 4, tanto la duración de la ventana que se quiere procesar como el intervalo de tiempo con el que se quiere realizar el procesado. Por ejemplo, se podría indicar que se quiere calcular la media en la última hora y realizar ese cálculo cada diez minutos.

Las transformaciones posibles con ventanas son las siguientes:

Transformación	Comportamiento
`window``(windowLength, slideInterval)`	Devuelve un DStream que es el resultado de aplicar la ventana (*window*) a los datos. El parámetro `windowLength` representa la duración de la ventana y, `slideInterval`, el desplazamiento de la ventana cada vez que se ha realizado un procesado.
`countByWindow``(windowLength, slideInterval)`	Devuelve el número de elementos que hay dentro de la ventana.
`reduceByWindow``(func, windowLength, slideInterval, [numTasks])`	Devuelve un DStream, que es el resultado de realizar una agregación definida en `func` con los elementos de la ventana.
`reduceByKeyAndWindow``(func, invFunc, windowLength, slideInterval, [numTasks])`	Realiza un `reduceByKey` para los elementos que hay dentro de la ventana.

Transformación	Comportamiento
countByValueAndWindow *(windowLength, slideInterval, [numTasks])*	Realiza un `countByValue` para los elementos que hay dentro de la ventana.

Debemos fijarnos que para tener el comportamiento de ventana fija (Tumbling window) hay que fijar los parámetros `windowLength` y `slideInterval` al mismo valor.

3.6. Las operaciones *join*

Spark Streaming también ofrece operaciones para trabajar mediante agregación y combinación de datos en *streaming* con datos estáticos, como se indica en el material teórico de este módulo. Estas operaciones, por facilidad de uso, están basadas en la sintaxis SQL. Las dos fundamentales son las siguientes:

1) **`Join stream-stream`**. Se realiza la combinación entre dos *streams* de datos. En el ejemplo, para cada intervalo *batch* se realizará el *join* entre el RDD correspondiente con el DStream de stream1 y el RDD correspondiente con el último intervalo del stream2.

```
streamCombinado = stream1.join(stream2)
```

2) **`Join stream-dataset`**. En este caso es necesario utilizar la función *transform* que se explicó en un apartado anterior.

```
dataset =... # es un RDD
windowedStream = stream.window(20)
joinedStream = windowedStream.transform(lambda rdd: rdd.join(dataset))
```

3.7. La persistencia

Antes de nada, es necesario recordar que en Spark existen dos tipos de métodos: las acciones y las transformaciones.

- Las **transformaciones** que se aplican a un RDD darían como resultado otro RDD y estas operaciones no se ejecutan en el momento de llamar a dicha transformación, sino en el momento de llamar a una acción. Es decir, las transformaciones, por naturaleza, se definen como *lazy*. Ejemplos de transformaciones serían `map`, `filter`, `groupByKey`, etc.

- Las **acciones** se aplican a los RDD y dan un resultado que no es un RDD. Se realizan todas las transformaciones indicadas para el RDD antes de aplicar la acción. Ejemplos de acciones serían `count()`, `collect()`, `reduce()`, etc.

De esta forma, cada vez que se aplique una acción a un RDD se ejecutarán todas las transformaciones asociadas, lo que aumentaría el tiempo de procesado si se llaman a varias acciones sobre el mismo RDD.

De la misma forma que se hace con los RDD, los DStreams también permiten que los datos persistan en memoria o en disco. Para ello, se utilizan los mismos métodos `cache()` y `persist()`, siendo útil cuando el DStream se va a utilizar para realizar varias operaciones con los datos, ya que permite guardar en memoria el resultado intermedio.

Para almacenar el DStream, por defecto, se utiliza la memoria, pero también se puede utilizar el disco duro o una combinación de ambos. Los DStreams generados por operaciones basadas en *windows* ya se almacenan de forma persistente, por lo que no es necesario llamar al método `persist()`.

3.8. *Checkpointing*

Se necesita que las aplicaciones de Spark Streaming funcionen de forma continuada y que sean tolerantes a los fallos. Esto se consigue utilizando el *checkpointing*.

El *checkpointing* consiste en almacenar el estado de funcionamiento en un sistema de almacenamiento confiable, como HDFS, para que se pueda recuperar en caso de fallo. Existen dos tipos de datos que se almacenan y son los siguientes:

- **Metadatos**. Hace referencia a la información de configuración para crear el *pipeline* de procesado (operaciones sobre el DStream que definen la aplicación) y los *micro-batches* que están en la cola de ejecución, pero todavía no se han lanzado.

- **Datos**. Serían los datos que contienen los RDD que forman parte del DStream y que se guardan para poder realizar operaciones *stateful* es decir, operaciones que dependen de *micro-batches* que se han ejecutado con anterioridad.

Hay importantes diferencias entre utilizar `persist` y `checkpoint`. Si se utiliza `persist()`, almacenando la información en disco con la opción `DISK_ONLY`, se almacena el RDD para que no sea necesario volver a calcularlo y, cuando se completa el *job*, se limpia la *cache* y se borra el disco.

El *checkpointing* almacena el RDD en HDFS. Borra las transformaciones que lo crearon. Completado el procesado, no se borra el archivo de *checkpoint*. Primero se guarda en *cache* antes de escribirlo en el directorio de *checkpointing*.

Resumiendo, con el *checkpointing* se consigue *fault tolerance*, cuando se necesita.

3.9. *Fault-tolerance semantics*

La semántica de los sistemas de *streaming* se clasifica en función de cuantas veces puede ser procesado por el sistema un mismo registro. Como ya se ha visto, puede haber tres tipos de garantías que se pueden proporcionar y son las siguientes:

- ***At most once***: cada registro se procesará una vez o ninguna.
- ***At least once***: cada registro se procesará al menos una vez. Garantiza que no se perderán datos, pero puede haber duplicados.
- ***Exactly once***: cada registro se procesará exactamente una vez. No se perderán datos ni serán procesados más de una vez. Es el más seguro de los tres.

En **cualquier proceso de *streaming*** habrá tres pasos para procesar los datos, que son los siguientes:

- **Recepción de los datos**: los datos se reciben normalmente de fuentes utilizando un conector específico.
- **Transformación de los datos**: los datos recibidos se transforman mediante operaciones.
- **Envío de los datos**: los datos transformados finales se envían a sistemas externos como sistemas de archivos, bases de datos, paneles de control, etc.

Para conseguir que se alcance una garantía *exactly once* de extremo a extremo es necesario que exista esta garantía en cada paso. Es decir, cada registro debe ser recibido *exactly once*, transformado *exactly once* y el envío de datos también se debe realizar *exactly once*.

En el caso de **Spark Streaming**, las garantías serían las siguientes:

- **Recepción de los datos:** la garantía depende de la fuente de datos.
- **Transformación de los datos:** con los RDD se garantiza que la transformación será *exactly once*.
- **Envío de los datos:** garantiza, al menos, una semántica *at least once* ya que depende de la operación de salida. Por programa se puede implementar una semántica *exactly once*.

Por tanto, si se quiere tener una garantía *exactly once* será necesario utilizar una fuente de datos que lo permita y programar el envío de datos para que lo sea.

3.10. Ejemplo básico

En este ejemplo se va revisar el ejemplo más clásico de procesado de datos en *streaming*. Podríamos llamarlo el «hola mundo» de *streaming*. Lo contextualizamos en la tecnología de Spark Streaming y veremos cómo podemos contar las veces que se recibe una palabra (*word count*), utilizando una entrada de datos mediante *sockets*. La forma de proceder es similar en todos los proyectos en los que se utilice Spark Streaming.

Primero debemos crear un `StreamingContext (sc)` a partir del `SparkContext, (sc)` pasando como parámetro el intervalo de tiempo para el procesado. En el ejemplo lo configuramos de un segundo. Es decir, el *micro-batch* va a consistir en todos los datos que se reciban en un segundo. Cabe remarcar que en este caso la ventana se asume fija (*tumbling window*).

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

# Crea un SparkContext local con 2 hilos
sc = SparkContext("local[2]", "CuentaPalabras")

# Crea un StreamingContext a partir del SparkContext con un tiempo de batch de 1 segundo
ssc = StreamingContext(sc, 1)
```

De forma similar al `SparkContext` en el procesado *batch*, el objeto `StreamingContext` que se ha creado será el que nos proporciona las funcionalidades de *streaming*.

A partir del `StreamingContext (ssc)` se crea un DStream (que llamaremos *líneas*). Como ya hemos visto Spark nos permite recolectar datos de varias fuentes. En este caso, como ya hemos dicho, vamos a hacerlo mediante un *socket* en el que asumimos que vamos a recibir texto. El *socket* se ejecutará en el propio ordenador (*localhost*) y en el puerto 9999.

```
# Crea un DStream que se conectará a "localhost" y puerto 9999 con una conexión de tipo socket
lineas = ssc.socketTextStream("localhost", 9999)
```

Una vez se tenga definida la entrada de datos mediante el DStream, definiremos el procesado, utilizando algunos de los métodos que ya conocemos de los RDD: `map`, `flatMap`, `reduceByKey`, etc. Recordemos que estos métodos se aplican ahora a un DStream, que tiene características distintas a los RDD. Por ejemplo, se ejecuta mediante *micro-batches* de ventana fija.

```
# Separa cada línea en palabras
palabras = lineas.flatMap(lambda linea: linea.split(" "))
#Transforma el DStream palabras en un DStream tuplas, con un valor de 1 para cada palabra
tuplas = palabras.map(lambda palabra: (palabra, 1))
```

```
#Aquí se contarán el número de palabras en modo StateLess
contadorPalabras = tuplas.reduceByKey(lambda x, y: x + y)
# Muestra por pantalla el número de veces que aparece cada palabra en el RDD
contadorPalabras.pprint()
```

Recordemos que queremos obtener el número de veces que ha aparecido cada palabra en el *stream*. Para ello, lo primero que hacemos es dividir las palabras de cada entrada de datos (`flatMap`), después mapeamos las palabras en pares «clave, valor» (`map`) y, finalmente, agregamos los contajes por clave (`reduceByKey`). Hasta este punto solo hemos definido las transformaciones y acciones que vamos a realizar. Pero, similar a los RDD en las operaciones *lazy* aún no se está ejecutando ninguna operación. Para poder comenzar el procesado es necesario llamar al método `start` de `StreamingContext`.

```
ssc.start() # Comienza el procesado
ssc.awaitTermination() # espera el fin del procesado
```

En este punto el proceso de *streaming* queda pendiente de recibir los datos.

Una vez definido este consumidor, vamos a ver cómo podemos definir un generador de datos. En este caso, por sencillez, lo vamos a hacer mediante el programa *netcat*. Para poder probar este ejemplo, previamente se debe ejecutar `netcat` y configurarlo como servidor de datos. Vamos a iniciarlo de la siguiente manera:

Netcat

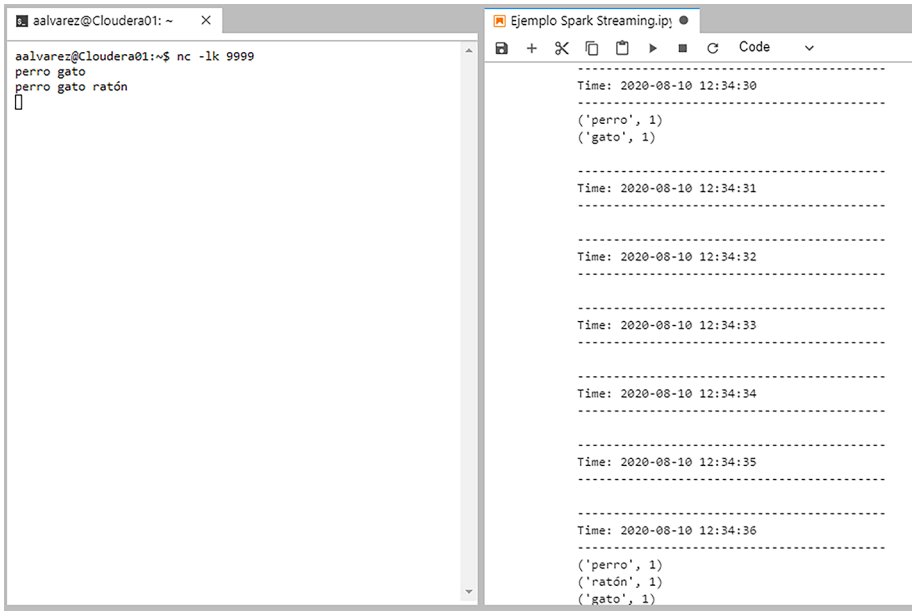
`Netcat (nc)` es una utilidad que permite escribir y leer desde conexiones de red utilizando TCP o UDP.

```
nc -lk 9999
```

Para que quede más claro lo que estamos haciendo, el parámetro `l` indica que queremos el `netcat` en modo escucha (*listen*). Es decir, va a esperar a que otros procesos se conecten a él en vez de iniciar la conexión. El parámetro `k` indica al `netcat` que no se cierre cuando el consumidor cierra la conexión. Va a estar a la espera de otras conexiones futuras. Finalmente, `9999` indica el puerto de escucha. Debemos fijarnos en que este puerto corresponde al mismo que hemos configurado en DStream.

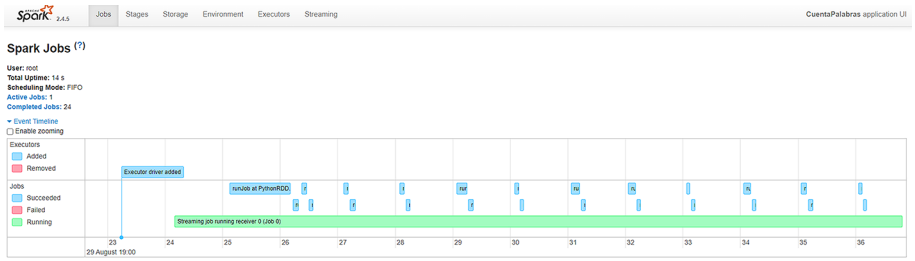
Finalmente, una vez ejecutado en un terminal, los datos que se sirvan serán las palabras que se escriban en dicho terminal, como se aprecia en la figura 5. Se puede observar que el procesado se realiza cada segundo, aunque no hayan llegado nuevas palabras.

Figura 5. Ejecución del ejemplo descrito en el subapartado 3.10



En el ejemplo, a la izquierda, ***netcat*** sirve palabras en el puerto 9999 y, a la derecha, Spark Streaming realiza el procesado cada segundo

Figura 6. Spark UI: los *jobs* de Spark ejecutándose para el programa anterior



Se puede comprobar en la figura 6 (muestra la ejecución de los diferentes *jobs* de Spark) que el cálculo se ha realizado únicamente para el DStream del intervalo configurado, en este caso, un segundo.

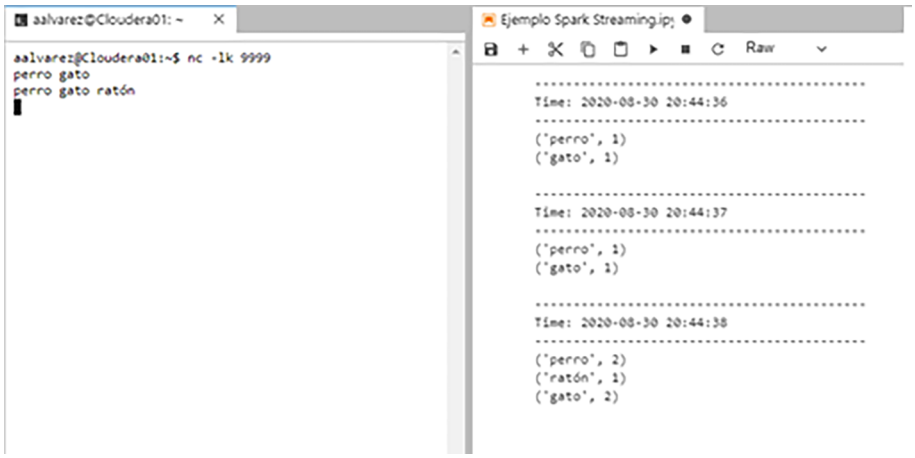
Esta forma de procesar los datos, como ya vimos, se llama *stateless* y es como funciona por defecto Spark Streaming. Si se quisiera poder realizar una aplicación que permita guardar los valores a lo largo del tiempo (*stateful*), se debería utilizar el método `updateStateByKey` que se explicó anteriormente.

```
def actualizaStateFul(nuevo, sumaAnterior):
return sum(nuevo) + (sumaAnterior or 0)
#Aquí se contarán el número de palabras en modo StateFul
contadorPalabrasStateFul = tuplas.updateStateByKey(actualizaStateFul)
```

Es necesario indicar previamente en qué carpeta se guardarán los valores previos, utilizando el método *checkpoint* con el objeto ssc (`StreamingContext`).

```
ssc.checkpoint("checkpoint")
```

Figura 7. Procesado *stateful*



4. Spark Structured Streaming

La API de Structured Streaming aparece en la versión 2 de Spark, que salió en el año 2016. Está integrado en Spark SQL y proporciona una forma más eficiente y cómoda de tratar los datos en flujo con Spark. Este modelo de *streaming* está basado en la API de *dataframe* y *dataset* (Spark SQL). Por tanto, se podrán realizar consultas SQL sobre los datos en *streaming*.

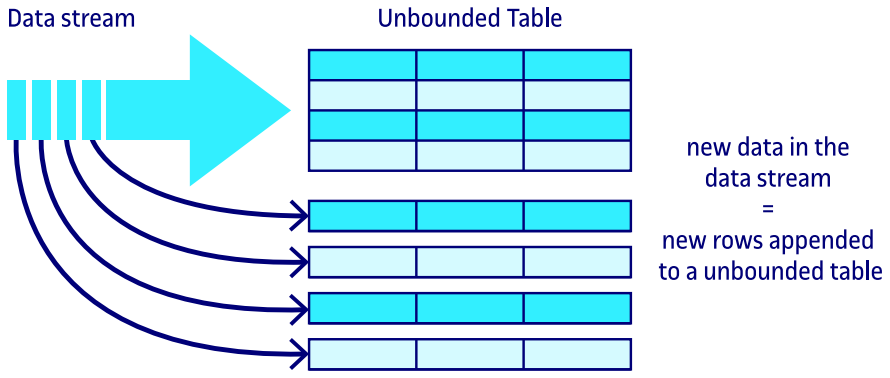
La ventaja más evidente es que permite procesar los datos de la misma forma que si se estuviera realizando un procesado por lotes con Spark SQL; es decir, se podría utilizar SQL o la API de *dataframe* para realizar el procesado de los datos (alto nivel). Pero, además de esta ventaja, ofrece otras como, por ejemplo, la posibilidad de trabajar con ***event-time windows***, la posibilidad de realizar ***join*** entre los datos procedentes de *streaming* (***stream-stream join***) o de los datos en *streaming* con datos estáticos (***stream-table join***) y también ofrece la posibilidad de realizar agregaciones con los datos que se estén recibiendo.

Además, proporciona mecanismos que permiten asegurar una tolerancia a los fallos ***end-to-end exactly once***.

4.1. Los conceptos básicos

Desde un punto de vista conceptual, la idea principal del Structured Streaming es tratar el flujo de datos (*data stream*) como una tabla ilimitada (*unbonded table*) a la que se le van añadiendo filas con los datos que se reciben, como se puede apreciar en la figura 8. De esta forma, se tiene un modelo muy similar al que se utiliza para el procesado por lotes. El procesado de los datos se escribe de la misma forma que se hace sobre una tabla de datos estática y Spark ejecuta la consulta periódicamente sobre dicha tabla ilimitada.

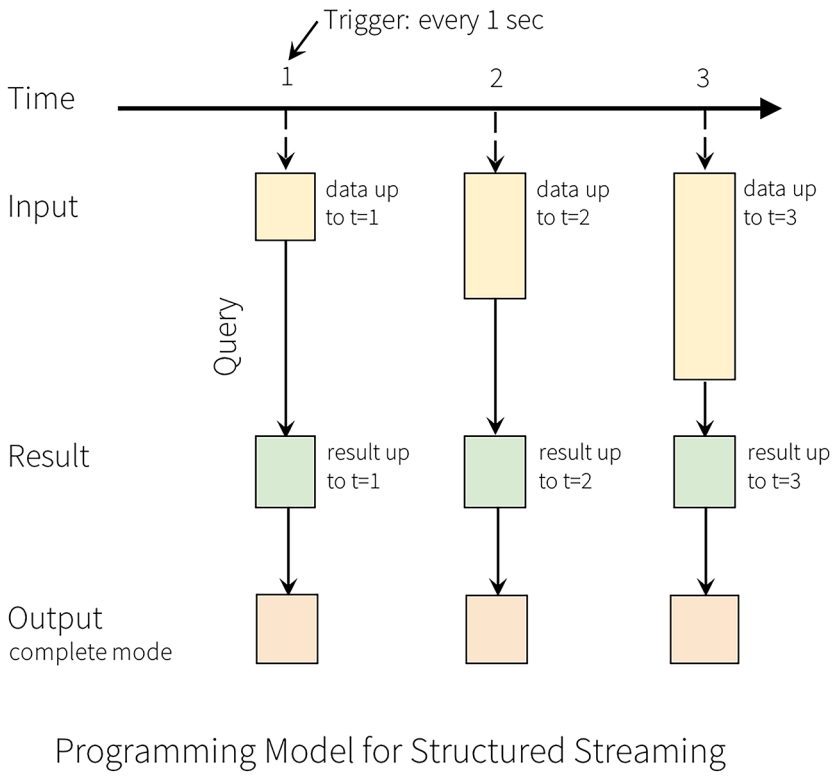
Figura 8. Dualidad de *stream*-tabla en Structured Streaming



Fuente: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

La consulta aplicada sobre los datos generará una tabla de resultado. Por ejemplo, si se actualiza la lectura de los datos de entrada cada segundo, se aplicará la consulta que, si procede, actualizará la salida. Cuando se actualice la tabla de resultado, se podrán escribir los cambios a un destino externo como, por ejemplo, una unidad de almacenamiento o un *topic* de Kafka.

Figura 9. *Pipeline* de procesado en Structured Streaming



Fuente: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

El Structured Streaming no almacena la tabla completa, sino que lo que hace es leer los últimos datos de la fuente de *streaming*, los procesa actualizando el resultado y descarta los datos originales. Es importante tener en cuenta que solo se almacenan los datos intermedios necesarios para poder actualizar el resultado.

Para poder configurar una consulta de *streaming* será necesario programar los siguientes pasos:

- Definir la fuente de entrada (*input source*).
- Transformar los datos recibidos.
- Definir la salida (*output sink* y *output mode*).
- Detallar cómo se procesarán los datos.
- Comenzar la consulta.

Es importante destacar que los cuatro primeros pasos son de configuración y es el último paso el que pone en marcha el procesado.

Antes de entrar a detallar cada uno de estos pasos, vamos a verlos con un ejemplo.

4.2. Ejemplo de Spark Structured Streaming

Al igual que en el caso de Spark Streaming, se va a analizar una aplicación para contar palabras que se reciben desde un servidor por medio de un *socket* TCP.

Inicialmente, como en cualquier programa en Spark SQL, se importan las librerías y se declara la variable `spark`.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split
spark = SparkSession \
.builder \
.appName("ContadorDePalabras") \
.getOrCreate()
```

Ahora se empezaría con los pasos necesarios para realizar el procesado:

1) Definir la fuente de entrada (*input source*). Se indica el tipo de fuente de entrada y los parámetros de configuración necesarios para establecer la conexión. En este caso, se indica que es una conexión de tipo `socket` a `localhost` (el ordenador donde se está ejecutando) y por el puerto 9999.

```
# Crea el DataFrame líneas que representa el flujo de palabras del servidor localhost:9999
lineas = spark \
.readStream \
.format("socket") \
.option("host", "localhost") \
.option("port", 9999) \
.load()
```

2) Transformar los datos recibidos. Se configuran las operaciones que se quieren realizar sobre el *data frame* que representa a los datos. En este caso, será extraer las palabras de cada una de las líneas y contar el número de veces que aparece cada palabra.

```
# con la función split se recuperan las palabras de cada línea en un array
# con la función explode cada elemento del array pasa a ser una nueva fila del DataFrame
palabras = lineas.select(
explode(
split(lineas.value, " ")
```

```
).alias("palabra")
)
# Agrupa por palabra y cuenta las veces que aparecen
cuentaPalabras = palabras.groupBy("palabra").count()
```

3) Definir la salida (*output sink* y *output mode*). En este paso se define el destino de los datos y cómo será el modo de salida (`complete`, `append` o `update`).

```
# Configura la salida por "console" y el modo de salida: "complete"
salida = cuentaPalabras.writeStream.format("console").outputMode("complete")
```

4) Detallar cómo se procesarán los datos. En esta etapa se indicará cada cuanto tiempo se lanzará el procesado para generar la salida mediante el método *trigger.* Además, también se puede configurar el *checkpoint.* El *checkpoint,* como ya se vio en Spark Streaming, consiste en indicar una carpeta en el sistema de archivos HDFS, en la que se guardarán los datos para conseguir una tolerancia a los fallos *exactly-once*.

```
# Configura que se procese cada segundo y una carpeta HDFS para recuperación de fallos
consulta = salida.trigger(processingTime="1 second")
.option("checkpointLocation", "/Carpeta_en_HDFS")
```

5) Comienzo de la consulta. Con el método `start` comenzaría a realizarse el procesado de los datos. Hasta este paso solo se estaba configurando el comportamiento deseado.

```
# Comienza la ejecución de la consulta que muestra el resultado por la consola
consulta.start()
```

Os recomendamos probar el ejemplo anterior sobre la máquina que tengáis disponible. Para hacerlo, previamente se debe ejecutar *netcat* y utilizarlo como servidor de datos.

Netcat

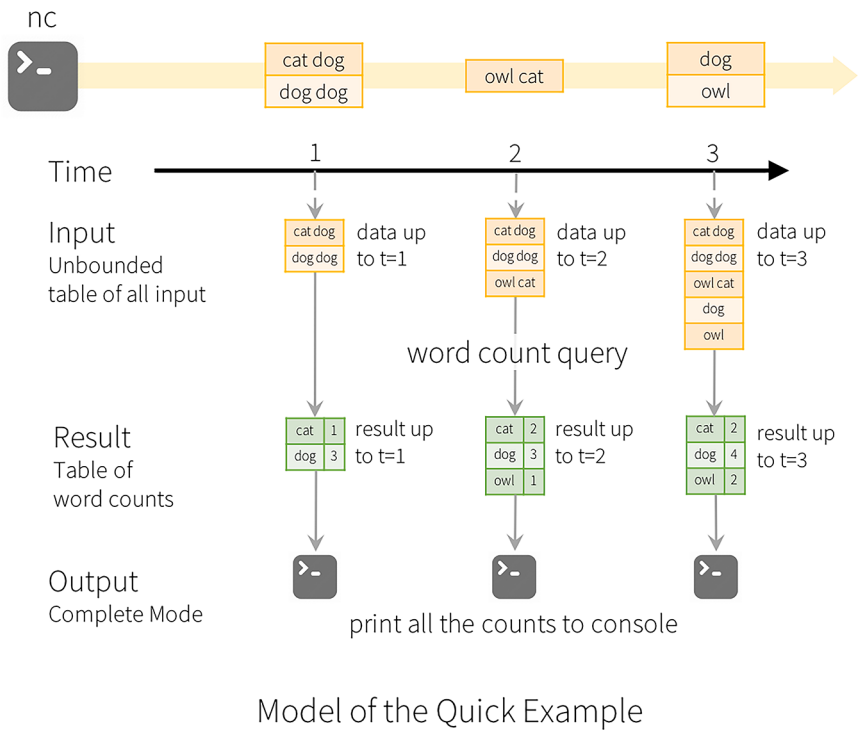
`Netcat (nc)` es una utilidad que permite escribir y leer desde conexiones de red utilizando TCP o UDP.

```
nc -lk 9999
```

Se ejecutará en otro terminal y los datos que se sirvan serán las palabras que se escriban en dicho terminal.

En la figura 10, se puede ver un gráfico del funcionamiento del programa. Ante los datos de entrada (*input*) se generaría un resultado (*result*) y se sacaría por pantalla (*console*).

Figura 10. Ejemplo ilustrativo de la ejecución de un *wordcount* en Structured Streaming



Fuente: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

4.3. Las fuentes de entrada

Apache Spark soporta, de forma nativa, la lectura de flujos de datos de Apache Kafka y de todos los tipos de archivos que se pueden leer con Spark SQL. Las fuentes soportadas son las siguientes:

1) Ficheros. Structured Streaming trata los archivos que se escriben en una carpeta como un flujo de datos (*data stream*). Se pueden leer los formatos de archivos que puede leer Spark SQL (JSON, CSV, parquet, etc.) y hay que tener en cuenta las siguientes restricciones:

- Todos los archivos que estén en la carpeta deben estar en el mismo formato y deben tener la misma estructura (*schema*).
- Los ficheros deben estar disponibles una única vez para una lectura completa y cuando ya están disponibles no pueden ser modificados. Es decir, los ficheros tienen que «aparecer» ya completos en el directorio y no se pueden ir añadiendo líneas a estos una vez que aparecen.

2) Kafka. Para conectarse a Kafka es necesario configurar las opciones para la conexión. En el siguiente ejemplo se puede ver cómo establecer la conexión con el *topic* `temperaturas`:

```
dfKafka = (spark
.readStream
```

```
  .format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("subscribe", "temperaturas")
  .load())
```

La estructura (*schema*) del *data frame* que se devuelve es la siguiente:

Columna	Tipo
key	binary
value	binary
topic	string
partition	Int
offset	Long
timestamp	timestamp
timestampType	Int
headers (optional)	array

3) Socket. Permite leer texto en codificación UTF8 desde una conexión *socket*. Solo debería utilizarse para realizar pruebas y no proporciona tolerancia a los fallos *exactly once*.

4) Rate. Genera datos creando un número previamente configurado de filas por segundo. La salida contiene dos columnas: *timestamp* y *value*. La columna *timestamp* contiene el momento en que se envió el mensaje y *value* contiene el número de mensaje enviado empezando por 0. Se utiliza para realizar pruebas y *benchmarking*. Se podría utilizar para realizar las pruebas antes de consumir de un flujo de datos real.

4.4. Las salidas de *streaming*

A la hora de determinar la salida que proporcionará la consulta tendremos que configurar de qué forma queremos que se escriba la tabla resultado en el almacenamiento externo (*output mode*) y también el formato y el destino de dicha tabla (*output sink*).

La salida, en función de cómo se escriba la tabla resultado en el almacenamiento externo, puede ser definida de diferentes modos, como vemos a continuación:

- **Modo *complete*.** En este modo, la tabla resultante se escribe entera.
- **Modo *append*.** Solo se añaden las filas nuevas de la tabla resultante desde el último *trigger*.

- **Modo *update***. En este caso, solo se añadirán las filas de la tabla resultante que se hayan modificado desde el último *trigger*.

Cada uno de estos modos se utilizará en función del procesado que se quiera realizar.

Los destinos soportados son los siguientes:

1) Ficheros. Structured Streaming permite escribir los formatos de archivos que puede escribir Spark SQL. Solo permite escribir a archivos en modo `append`, ya que en este modo se añaden archivos a la carpeta. No se permiten los modos `complete` y `update`.

2) Kafka. Para escribir hacia Kafka, Structured Streaming espera que el *dataframe* tenga la estructura que se muestra en la siguiente tabla:

Columna	Tipo
`key (optional)`	`string or binary`
`value (required)`	`string or binary`
`headers (optional)`	`Array`
`topic (optional)`	`String`
`partition (optional)`	`Int`

3) Consola (para depurar). Cada vez que hay un *trigger* saca por la pantalla la tabla resultado. Se utiliza para depurar errores.

4) Memoria (para depurar). Cada vez que hay un *trigger* guarda en memoria la tabla resultado. Admite los tres modos: *complete, append* y *update*.

5) Personalizable. Utilizando los métodos `foreach()` y `foreachBatch()`, se pueden ejecutar las funciones que permitan escribir el resultado hacia otros destinos diferentes como Cassandra, MongoDB, etc.

4.5. Las transformaciones

La gran mayoría de las transformaciones que existen en Spark SQL se van a poder aplicar en *Structured Streaming*. Las transformaciones se pueden clasificar en dos grupos:

1) Transformaciones *Stateless*. Son aquellas transformaciones que no requieren información de otras filas, como por ejemplo, `select()`, `filter()` o `map()`. Se van a poder ejecutar en Structured Streaming y los únicos modos aplicables son `append` y `update`.

2) Transformaciones *Stateful*. Son transformaciones que requieren conocer el estado de otras filas, como por ejemplo, `groupBy()`. Algunas de estas transformaciones no se pueden aplicar y mostrarán un mensaje «operation XYZ is not supported with streaming DataFrame». Las que se aplicarán sin problema son las agregaciones que no están basadas en el tiempo y las de ventanas con *event time*.

4.6. Las ventanas con *event time* y *watermarking*

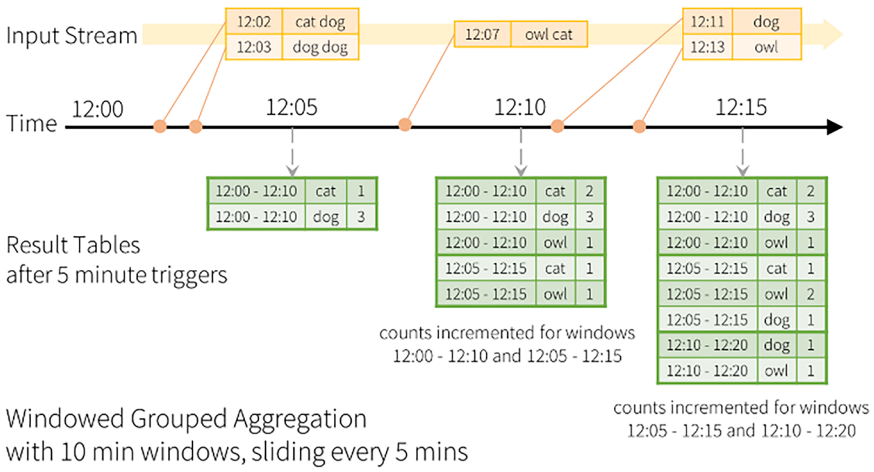
Normalmente, las agregaciones que se utilizarán en *streaming*, se realizan únicamente sobre una determinada ventana temporal y no a lo largo del tiempo.

Para entender el funcionamiento de las ventanas vamos a utilizar el ejemplo ya conocido del contador de palabras. En este caso, como se puede comprobar en la figura 11, cada línea con palabras llega en un momento diferente y hay una columna *timestamp* que contiene el momento en que se envía el mensaje; es decir, el momento en que se genera el evento (*event time*).

En la línea temporal, se ve el instante en el que llega la información y, en este caso, por simplicidad, coincide con el instante en el que se produjo el evento (*event time*).

El *trigger* para procesar los datos se lanza cada cinco minutos y se pretende generar una tabla que cuente las palabras que se han generado en los últimos diez minutos antes del *trigger*.

Figura 11. Ejemplo de ejecución de un *trigger* para relizar un *wordcount* en Structured Streaming



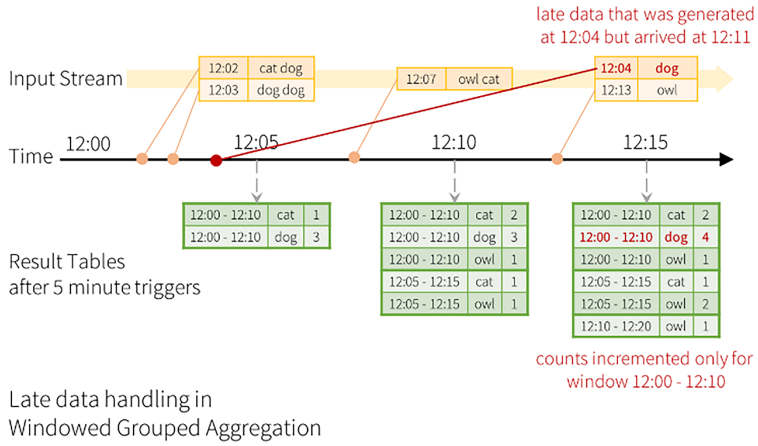
Fuente: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Como ya se vio anteriormente, es importante la gestión de la información que llega con retardo.

Así que vamos a considerar que un dato que se generó a las 12:04 llegó a las 12:11. Se observa en la figura 12 que, en este caso, se actualizará la agregación correspondiente al

momento temporal en que se produjo el evento; es decir, la comprendida entre las 12:00 y 12:10, cuando se recalcula a las 12:15.

Figura 12. Ejemplo de ejecución de un *trigger* para relizar un *wordcount* con eventos retardados



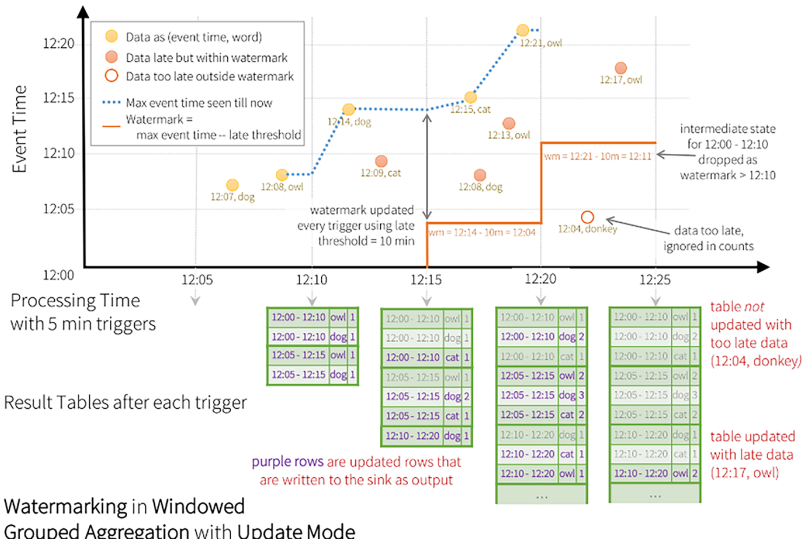
Fuente: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Para poder trabajar con datos retardados, Structured Streaming necesita almacenar los valores intermedios de las agregaciones que se han realizado. Se debe recordar que no se almacenan todos los valores que llegan, ya que eso podría saturar el medio de almacenamiento, sino que se almacenan los valores intermedios de las agregaciones.

Esta forma de trabajar permite que se gestionen los valores que llegan con retardo, pero para evitar que esos valores ocupen mucho espacio y optimizar tiempo de cálculo, se definen los *watermarking*. Estos *watermarking* permiten definir cuando se dejan de tener en cuenta estos valores que llegan con retardo.

El ejemplo ilustrativo que se muestra en figura 13 lanza el *trigger* para procesar los datos cada cinco minutos para los diez minutos anteriores, descartando aquellos valores que lleguen con más de diez minutos adicionales de retardo.

Figura 13. Ejemplo del uso de *watermarks* en Structured Streaming



Fuente: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

El programa para implementar el ejemplo de la figura 13 sería el siguiente:

```
words =... # streaming con la siguiente estructura: { timestamp: Timestamp, word: String }
# Agrupa los datos por window y word, procesando la cuenta para cada grupo
windowedCounts = words \
.withWatermark("timestamp", "10 minutes") \
.groupBy(
window(words.timestamp, "10 minutes", "5 minutes"),
words.word) \
.count()
```

4.7. Las operaciones *join*

Se pueden realizar operaciones *join* entre datos en *streaming* (*streaming-streaming*) o de datos estáticos con datos en *streaming* (*streaming-dataset*), como ya se vio en el material teórico de este módulo.

1) *Join stream-static*. Permite el *inner join*, el *left outer join* (si el *stream* es el que llama al método `.join`) y el *right outer join* (si es el *dataset* el que llama al método `.join`)

```
staticDf = spark.read.... # Aquí se configura el Dataframe estático con el que se realiza el join
streamingDf = spark.readStream.... # Aquí se configura el Streaming
streamingDf.join(staticDf, "type")
```

2) *Join stream-stream*. Se realiza la combinación entre dos *streams* de datos. La dificultad estriba en que cada uno de ellos está incompleto en el momento de hacer el *join*. También se puede utilizar el *watermarking* para descartar los datos que llegan con mucho retardo. Se pueden dar dos posibilidades de *join*:

a) *Inner join* con *watermarking* opcional. La información de cada uno de los *streams* de datos crece indefinidamente, por lo que será necesario limitar el *stream* resultante mediante una de las siguientes opciones:

- Utilizando el *watermarking*.
- Definiendo condiciones en función del tiempo para la realización del *join*.

b) *Outer join* con *watermarking*. Con los *outer join* es necesario definir el *watermarking* y, al mismo tiempo, incluir las condiciones temporales.

Ejemplo

Para ilustrar este tipo de *join* se presenta el siguiente ejemplo. Se tiene un caso en el que se pretende hacer el *join* entre las impresiones de un *banner* y los clics que se han realizado en dicho *banner*. Además de haber definido previamente los *streams* con *watermark*, se incluyen condiciones como la coincidencia de identificador (clickAdId = impressionAdId), que el clic sea posterior a la aparición del *banner* (clickTime >= impressionTime), pero se realice en menos de una hora (clickTime <= impressionTime + interval 1 hour).

```
impressionsWithWatermark.join(
clicksWithWatermark,
expr("""
clickAdId = impressionAdId AND
clickTime >= impressionTime AND
clickTime <= impressionTime + interval 1 hour
"""),
"leftOuter" # puede ser "inner", "leftOuter", "rightOuter"
```

4.8. El caso de uso

Spark Structured Streaming se utiliza para múltiples aplicaciones ya que, como ya se vio, permite combinar datos de diferentes fuentes, tanto de fuentes en *stream* con fuentes de datos estáticas como de datos en *streaming* entre sí. Al utilizar las mismas estructuras que se utilizan en otros módulos de Spark permite integrarse fácilmente con ellos, como con MLlib (módulo de *machine learning* en Spark) para poder aplicar modelos predictivos. Por ejemplo para las siguientes funciones:

- análisis de las ventas en tiempo real para la gestión de *stocks* y la compra de materia prima
- comportamiento del cliente
- impresiones de anuncios en Internet y *clicks* sobre los anuncios
- ventas de productos en tiendas de moda
- temperaturas en diferentes ciudades

Cuando se están realizando pruebas, antes de disponer de los flujos de datos con los que se va a trabajar en producción, existen varias posibilidades. Una de ellas sería utilizar como fuente de entrada `rate`, como ya se indicó en el subapartado 4.3. A continuación, se muestra un posible ejemplo de uso para realizar esto:

Imaginemos que se pretende leer datos de un flujo que contendrá nombres de ciudades y temperaturas. Para poder probar el código que queremos programar, se puede utilizar `rate` para disponer de un flujo de datos simulado y modificar dicho flujo para que tenga la estructura similar al flujo real que se leerá en producción.

La definición de la fuente de entrada

Al definir la fuente de entrada se escoge `rate` que generará un *stream* con dos columnas: `timestamp` y `value`. La opción `rowsPerSecond` indica las filas que se generarán en cada segundo. La columna *value* se incrementará en 1 de una fila a la siguiente.

```
serie = (spark
   .readStream
   .format("rate")
   .option("rowsPerSecond", 1)
   .load())
```

Este *data frame* que se ha generado y que se actualiza cada segundo, debemos modificarlo para que tenga la misma estructura que el flujo real de datos. Para ello, debemos añadirle las mismas columnas que tendría dicho flujo, a partir de las columnas `timestamp` y `value` generadas.

En este caso, se ha modificado la columna *timestamp* para introducir fluctuaciones en el tiempo en función del valor del resto de dividir por 3, aunque se podría haber añadido aleatoriedad añadiendo la función `rand` de Spark. También se han generado de forma aleatoria (realmente en función del resto de dividir por diferentes números), una columna `Ciudades` y una columna `Temperatura`. Los valores asignados son el nombre de dos ciudades y un rango de temperaturas para la columna `Temperatura`.

```
# simula un timestamp con retardos y valores de temperatura variable que llegan de 2 ciudades
simula=serie.select(to_timestamp(unix_timestamp("timestamp")+100*col("value")%3).alias
("timestamp"),when(col("value")%3==0,"Vigo").otherwise("Barcelona").alias("Ciudades"),
(15+col("value")%5).alias("Temperatura"))
```

De esta forma, cada segundo, se añadirían nuevos datos al *dataframe* como los de la figura 14:

Figura 14. Ejemplo de ejecución de un programa que genera un flujo de datos mediante la operación

```
+-------------------+---------+-----------+
|          timestamp| Ciudades|Temperatura|
+-------------------+---------+-----------+
|2020-06-28 20:15:59|Barcelona|         16|
|2020-06-28 20:16:01|Barcelona|         15|
|2020-06-28 20:16:00|     Vigo|         16|
+-------------------+---------+-----------+
```

Para ver los valores que se generan en la simulación, se ha utilizado el siguiente código:

```
# para ver los valores simulados
valores = simula \
   .writeStream \
   .outputMode("append") \
   .format("console") \
   .start()
```

La transformación de los datos recibidos

El siguiente código propone un procesado de los datos para calcular la media de la temperatura de todas las ciudades que aparecen en el *stream*:

```
procesa=simula.groupBy("Ciudades").mean("Temperatura")
```

El inicio de la consulta

Para finalizar, se indica el `output mode` como `complete` y se define la salida (*output sink*) a `console`.

```
consulta = procesa \
   .writeStream \
   .outputMode("complete") \
   .format("console") \
   .start()
```

Es importante que el lector se fije en que para la definición de este ejemplo hemos seguido los pasos definidos en el subapartado 4.1. Este es el procedimiento recomendado para implementar *pipelines* de procesado con Structured Streaming.

Figura 15. En el resultado se muestran los datos simulados y los datos procesados

```
+-------------------+--------+-----------+
|          timestamp|Ciudades|Temperatura|
+-------------------+--------+-----------+
|2020-06-28 20:15:57|    Vigo|         15|
+-------------------+--------+-----------+

-------------------------------------------
Batch: 5
-------------------------------------------
+-------------------+---------+-----------+
|          timestamp| Ciudades|Temperatura|
+-------------------+---------+-----------+
|2020-06-28 20:15:59|Barcelona|         16|
|2020-06-28 20:16:01|Barcelona|         15|
|2020-06-28 20:16:00|     Vigo|         16|
+-------------------+---------+-----------+

-------------------------------------------
Batch: 6
-------------------------------------------
-------------------------------------------
Batch: 1
-------------------------------------------
+-------------------+---------+-----------+
|          timestamp| Ciudades|Temperatura|
+-------------------+---------+-----------+
|2020-06-28 20:16:02|Barcelona|         15|
+-------------------+---------+-----------+

+---------+----------------+
| Ciudades|avg(Temperatura)|
+---------+----------------+
|Barcelona|            15.5|
|     Vigo|            15.5|
+---------+----------------+

-------------------------------------------
Batch: 7
-------------------------------------------
+-------------------+---------+-----------+
|          timestamp| Ciudades|Temperatura|
+-------------------+---------+-----------+
|2020-06-28 20:16:04|Barcelona|         16|
|2020-06-28 20:16:03|     Vigo|         15|
|2020-06-28 20:16:05|Barcelona|         16|
+-------------------+---------+-----------+
```

5. Apache Storm

Apache Storm es un sistema distribuido de procesado en tiempo real. Está diseñado para procesar grandes cantidades de flujos de datos. Twitter adquirió Storm a su creador, Nathan Marz, y luego lo liberó como *open source* en 2011. Desde entonces, ha sido adoptado ampliamente por numerosas empresas.

5.1. La arquitectura: los componentes de Storm

Para entender el paralelismo en la arquitectura de Storm, es necesario entender que en un clúster Storm existen dos tipos de nodos: los nodos *master* y los nodos *worker*. El **nodo *master*** ejecuta un Daemon llamado **Nimbus**, que es el responsable de distribuir el código dentro del clúster, asignando tareas a cada nodo *worker*. Los **nodos *worker***, ejecutan un Daemon llamado **Supervisor**, que ejecuta una porción de una topología (un *pipeline* de procesado). Las topologías en Storm se ejecutan en varios nodos *worker* en diferentes máquinas.

Un clúster Storm sigue un modelo maestro-esclavo, en el que los procesos maestro y esclavo son coordinados por **Zookeeper**. A continuación, se describen con más detalle estos componentes:

1) Nimbus. Es el nodo maestro en un clúster Storm. Es el responsable de distribuir el código de la aplicación entre varios nodos *worker*, asignando tareas a las diferentes máquinas, monitorizando tareas en caso de fallo y reiniciándolas cuando sea necesario.

Nimbus es *stateless* y almacena todos sus datos en Zookeeper. Solo hay un único Nimbus en un clúster, pero está diseñado para que, en caso de fallo, se reinicie sin afectar a las tareas que se están ejecutando en los nodos *worker*.

2) Nodos *supervisor*. Son los nodos *worker* en un clúster Storm. Cada nodo supervisor ejecuta un Daemon que es el responsable de crear, iniciar y parar los procesos *worker* que ejecutan las tareas asignadas a ese nodo. Al igual que Nimbus, los *supervisor* almacenan la información relativa a su estado en Zookeeper, por lo que puede reiniciarse sin pérdida de información.

3) Zookeeper En las aplicaciones distribuidas, varios procesos necesitan coordinarse unos con otros y compartir información de configuración.

5.2. El modelo de datos de Storm

Storm permite procesar flujos de datos de forma distribuida y tolerante a los fallos. Para ello, se utilizan diferentes primitivas, cada una responsable de una tarea específica. Dichas primitivas se describen a continuación:

1) *Spout*. Fuentes de datos para la topología, por ejemplo, Kafka, Postgres, etc. Gestiona el flujo de entrada de datos (*input stream*) a un clúster Storm, para posteriormente pasar los datos que recibe a un componente llamado *bolt*.

Figura 16.
Spout

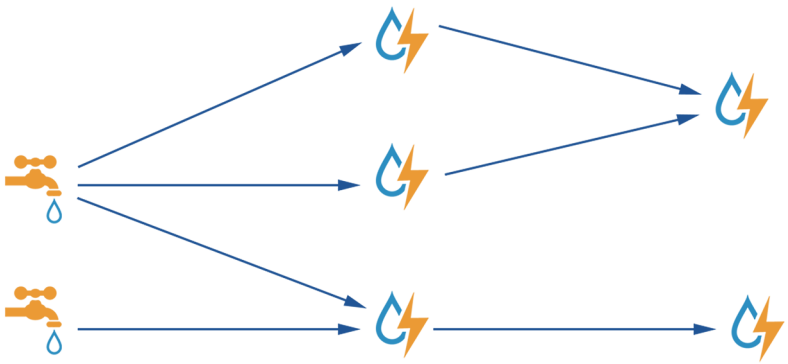
2) *Bolt*. Representa la unidad mínima de computación. Se encarga de realizar alguna transformación en los datos y, posteriormente, pasarlos a otro *bolt* o, sino, hace que los datos persistan, almacenándolos.

Es en esta unidad computacional donde se transforman los datos, realizando filtrado, agregación, transformaciones o *join*. Se pueden recibir datos de varias fuentes simultáneamente.

Figura 17.
Bolt

3) Topología. La forma en la que se organizan los *spouts* y los *bolts* se llama *topología*. Es un tipo de *directed acyclic graph* (DAG) que une los *spouts* y los *bolts*. Primero será necesario definir la topología y, posteriormente, programar los *bolts* y los *spouts*.

Figura 18. Ejemplo de topología en Storm



Fuente: https://storm.apache.org/

4) Tupla. Unidad de los datos que está sirviendo el Stream, en forma de una lista ordenada e inmutable de elementos.

Los *bolts* reciben tuplas (son inmutables) y emiten una tupla consistente en una modificación de la que han recibido. No se modifica la original ya que las tuplas son inmutables, sino que se emite la nueva con las transformaciones aplicadas.

5) Stream. Es una secuencia ilimitada de tuplas. En la topología, los vértices están constituidos por *spouts* y *bolts*. Los enlaces (*edges*) unen los vértices y están definidos por varios tipos de agrupamiento. En este caso, se presentan a continuación los cuatro tipos de enlaces más habituales:

- ***Shuffle grouping***. Distribuye las tuplas al siguiente *bolt* de forma aleatoria.
- ***Fields grouping***. Agrupa las tuplas en función del valor de una o varias columnas.
- ***All grouping***. Replica las tuplas en las siguientes etapas del *bolt*.
- ***Global grouping***. Envía todas las tuplas al siguiente *bolt*.

5.3. La implementación

Storm permite utilizar varios lenguajes de programación, aunque al estar realizado en Java, cuando se utilizan otros lenguajes se ejecutarán como subprocesos. En este curso se pretende realizar el procesamiento de los *streamings* utilizando Python, por lo que se podría optar por las siguientes opciones:

1) Se podría definir la topología en Java, referenciar en Java qué archivos de Python se corresponden con los diferentes *spouts* y *bolts* y, luego, programar los *spouts* y los *bolts* en Python.

2) Se podría utilizar el *framework* Flux que permitirá definir la topología mediante un archivo de configuración YAML y, luego, programar los *spouts* y los *bolts* en Python.

3) Se podría utilizar *streamparse*, que se basa en Thrift y permite definir también las topologías en Python.

La segunda opción es la que se adoptará en el curso por su simplicidad.

El archivo YAML de configuración de la topología tendrá los siguientes apartados:

- ***name***: nombre de la topología
- ***config***: parámetros de configuración de la topología
- ***spouts***: definición de los *spouts*
- ***bolts***: definición de los *bolts*
- ***streams***: enlace de los *spouts* y los *bolts* para definir una topología

A continuación, se va a presentar el ejemplo de un programa para contabilizar las apariciones de cada palabra, de forma similar a lo que se realizó con *spark streaming* y Structured Streaming, utilizando el *framework* Flux. Para ello, se dispondrá de un archivo de configuración YAML (*topology.yaml*), un archivo que contiene el *spout* (*sentencespout.py*) y dos archivos *bolt* (*splitbolt.py* y *countbolt.py*) para realizar la separación de palabras y la cuenta de estas.

A continuación, se verá el archivo *topology.yaml*:

```
name: "cuentaPalabras"
config:
   # Se indican el número de workers que se utilizarán
   topology.workers: 1
# Declaración de Spout
spouts:
   - id: "sentence-spout"
   className: "org.apache.storm.flux.wrappers.spouts.FluxShellSpout"
   constructorArgs:
      # Comando a ejecutar
      - ["python", "sentencespout.py"]
      # Campos de salida del spout
      - ["sentence"]
   # paralelismo
   parallelism: 1
# Declaraciones Bolt
bolts:
   - id: "splitter-bolt"
   className: "org.apache.storm.flux.wrappers.bolts.FluxShellBolt"
   constructorArgs:
      # Comando a ejecutar
      - ["python", "splitbolt.py"]
      # Campos de salida del bolt
      - ["word"]
   parallelism: 1
    - id: "counter-bolt"
   className: "org.apache.storm.flux.wrappers.bolts.FluxShellBolt"
   constructorArgs:
      # Comando a ejecutar
      - ["python", "countbolt.py"]
      # Campos de salida del bolt
      - ["word","count"]
   parallelism: 1
 # Logging
 - id: "log"
   className: "org.apache.storm.flux.wrappers.bolts.LogInfoBolt"
   parallelism: 1
# Declaraciones de Stream
```

```
streams:
   - name: "Spout --> Splitter" # sólo es informativo
   # Stream emisor
   from: "sentence-spout"
   # Stream consumidor
   to: "splitter-bolt"
   # Agrupamiento
   grouping:
      type: SHUFFLE
   - name: "Splitter -> Counter"
   from: "splitter-bolt"
   to: "counter-bolt"
   grouping:
      type: FIELDS
      # Campos por los que se agrupa
      args: ["word"]
   - name: "Counter -> Log"
   from: "counter-bolt"
   to: "log"
   grouping:
      type: SHUFFLE
```

En este archivo de configuración *topology.yaml* se indica el número de *workers* que se utilizarán y se definen los elementos de la topología, así como sus relaciones.

El siguiente archivo que vamos a ver es el *sentencespout.py*:

```
import storm
import random
# Frases que se enviarán
SENTENCES = """
Es importante entender que los datos de la Speed Layer son temporales, estos mismos datos
también son almacenados por la Batch Layer, con lo que, si algún problema ocurre en ella,
el sistema no queda comprometido porque el Batch Layer ya ha incorporado dichos datos al
sistema y, aunque con mayor latencia, estarán disponibles una vez las vistas Batch se
hayan recalculado. La complejidad de la Speed Layer es mucho mayor que la de la Batch ya
que debe utilizar algoritmia y lógica para hacer vistas incrementales, sin embargo, el
componente más complejo de la arquitectura es aquel que no compromete a todo el sistema.
""".strip().split('\n')
class SentenceSpout(storm.Spout):
   def initialize(self, conf, context):
      self._conf = conf
      self._context = context
         storm.logInfo("Iniciando instancia de SentenceSpout...")
   # Procesa la siguiente tupla
   def nextTuple(self):
```

```
        # envia una frase de forma aleatoria
        sentence = random.choice(SENTENCES)
        storm.logInfo("Emiting %s" % sentence)
        storm.emit([sentence])
# Inicia el spout cuando se invoca
SentenceSpout().run()
```

En este programa en Python, se configura el *spout* que va a enviar las frases. En este caso, las frases están contenidas en el propio *spout*.

splitbolt.py

En este *bolt* programado en Python se extraen las palabras de las frases que llegan:

```
import storm
class SplitBolt(storm.BasicBolt):
   def initialize(self, conf, context):
      self._conf = conf
      self._context = context
      storm.logInfo("Iniciando instancia de SplitBolt...")
   def process(self, tup):
      # Corta por los espacios las frases que están llegando
      words = tup.values[0].split()
      # Recorre words y envía cada palabra
      for word in words:
         storm.logInfo("Enviando %s" % word)
         storm.emit([word])
# Inicia el bolt cuando se invoca
SplitBolt().run()
```

countbolt.py

En este último *bolt* se cuentan las palabras:

```
import storm
from collections import Counter
class CountBolt(storm.BasicBolt):
   # Initialize this instance
   def initialize(self, conf, context):
      self._conf = conf
      self._context = context
       # Crea un nuevo contador para la instancia
      self._counter = Counter()
      storm.logInfo("Iniciando instancia de CountBolt...")
    def process(self, tup):
      # Recupera la palabra de la tupla que recibe
```

```
        word = tup.values[0]
        # Incrementa el contador
        self._counter[word] +=1
        count = self._counter[word]
        storm.logInfo("Enviando %s:%s" % (word, count))
        # emite la palabra y cuenta
        storm.emit([word, count])
# # Inicia el bolt cuando se invoca
CountBolt().run()
```

Bibliografía

Psaltis, Andrew G. (2017). *Streaming Data: Understanding the Real-Time Pipeline*. Manning Publications.

Spark Streaming Programming Guide <http://spark.apache.org/docs/latest/streaming-programming-guide.html>

Structured Streaming Programming Guide <http://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#structured-streaming-programming-guide>

Apache Storm Documentation <http://storm.apache.org/releases/2.2.0/index.html>