

**Componentes de arquitectura en
streaming**
© EDICIONES ROBLE, S.L.

Indice

Componentes de arquitectura en streaming	3
I. Introducción	3
II. Objetivos	3
III. Procesamiento de datos con Spark Streaming	3
3.1. Introducción	3
3.2. Conceptos	5
3.3. Un ejemplo simple	6
3.4. Arquitectura interna	11
3.5. Transformaciones	14
3.5.1. Stateless transformations	15
3.5.2. Stateful transformations	15
3.6. Operaciones de salida	17
3.7. Integración con fuentes de datos	18
3.8. Garantías de procesamiento	19
3.9. Consideraciones de rendimiento	20
IV. Resumen	20
Ejercicios	22
Caso práctico	22
Ejercicio 1	22
Ejercicio 2	22
Ejercicio 3	22
Recursos	24
Enlaces de Interés	24
Glosario.	24

Componentes de arquitectura en streaming

I. Introducción

En esta unidad, se presentará el *framework* de referencia para procesamiento de *streams*: Spark Streaming.

Spark Streaming es la especialización de Spark para trabajar con datos en tránsito. La ventaja fundamental de Spark Streaming es que las técnicas de procesamiento en *batch* que se aprenden y ensayan con Spark son perfectamente válidas para aplicarlas en *streaming*.

II. Objetivos



Al término de esta unidad, los alumnos habrán alcanzado los siguientes objetivos:

- Comprender la arquitectura de procesamiento que provee Spark Streaming para datos en tiempo real.
- Escribir procesamientos sencillos de *streams* de datos usando *notebooks* en el servicio *cloud* de Spark de Databricks.

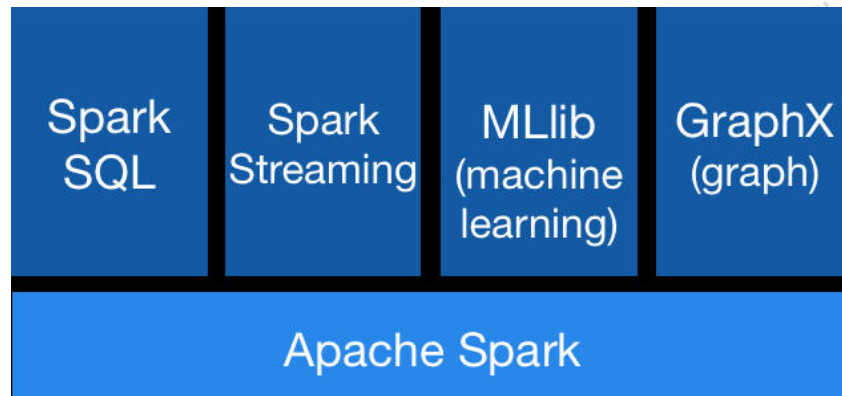
III. Procesamiento de datos con Spark Streaming

3.1. Introducción

Spark Streaming¹ se añadió en 2013 como extensión al API del *core* y es el módulo de Spark que nos permite procesar datos en tránsito y en *near-real time*. Con este módulo, podemos desarrollar aplicaciones de procesamiento de *streams* usando un API muy similar al que usaríamos para procesamiento *batch* reutilizando no solo conocimiento, sino incluso código fuente.

¹Página web de Apache Spark Streaming. [En línea] URL disponible en: <https://spark.apache.org/streaming/>

Figura 1. Documentación de Spark. Módulos del framework de Spark.



Fuente: <https://spark.apache.org/>

En general, Spark proporciona un ecosistema unificado para diferentes necesidades de procesamiento y eso ha sido clave para la rápida adopción de Spark Streaming.

Esa total integración entre los módulos del ecosistema de Spark nos permite implementar arquitecturas complejas en las que procesar el *stream* de datos usando Spark SQL y DataFrames, así como aplicar técnicas de *machine learning* con MLlib sin salirnos de lo proporcionado por el *framework*. De este modo, es posible implementar aplicaciones de *scoring* y aprendizaje automático *online* y continuo. O combinar fuentes de datos estáticas externas que enriquezcan el *stream* de manera sencilla usando SQL.

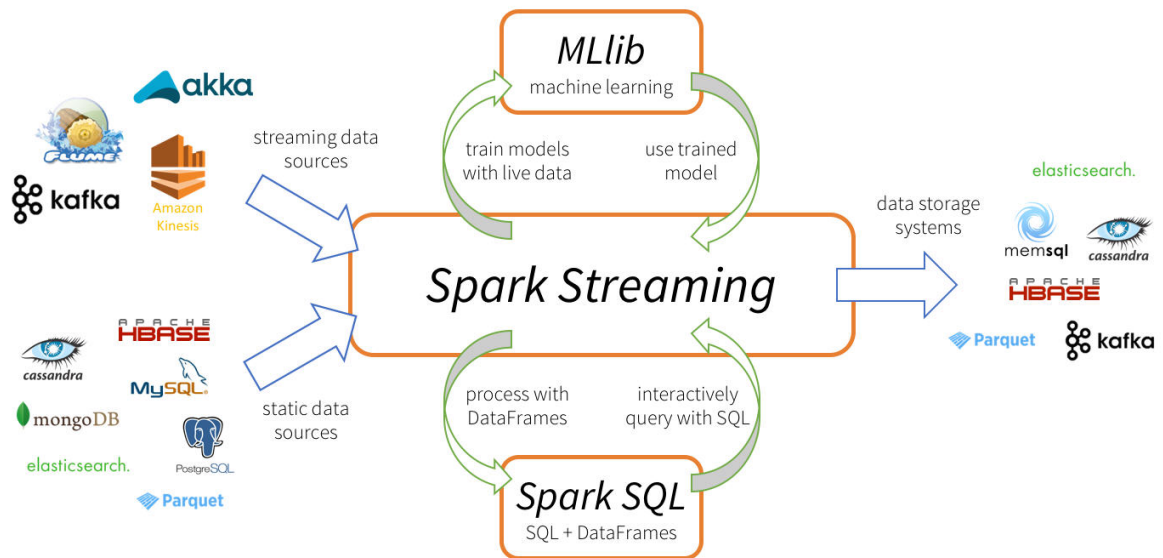


Figura 2. Uso de diferentes módulos de Spark en procesamiento de streams.

Fuente: Datanami.com, <https://www.datanami.com/2015/11/30/spark-streaming-what-is-it-and-whos-using-it/>

Los casos de uso más habituales de Spark Streaming hoy en día son los siguientes:

ETL continuo

Para limpiar y agregar datos en tránsito antes de almacenarlos en almacenamiento persistente.

Enriquecimiento de datos

Los datos del *stream* se combinan en tránsito con fuentes de datos externas usando *joins* para dar lugar a análisis *online* más completos.

Detección de eventos o situaciones relevantes

Por ejemplo, la identificación de comportamientos anómalos que deben desencadenar las acciones correspondientes.

Análisis continuo de comportamiento online

En el caso de aplicaciones web o móviles, es lo que se conoce como *sessionization*.

3.2. Conceptos

Así como Spark está construido alrededor del concepto de RDD, Spark Streaming proporciona una abstracción denominada DStreams o *discretized stream*.

Un DStream es una secuencia de datos que llegan a lo largo del tiempo. Internamente cada DStream es una secuencia de RDDs. Es decir, Spark Streaming “trocea” el *stream* a intervalos regulares de tiempo.



Figura 3. Discretized stream en Spark Streaming.

Fuente: Elaboración propia.

El modelo de procesamiento a muy alto nivel está representado en la figura 4. Spark Streaming recibe los datos del *stream* y los divide en *batches*. Cada uno de esos *batches* los procesa el motor de Spark, que genera un nuevo *stream* de salida con los resultados del procesamiento. El *stream* de salida estará formado entonces por una secuencia de *batches* de datos procesados.



Figura 4. Documentación de Spark. Modelo de procesamiento de microbatches en Spark Streaming.

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

Los DStreams se pueden crear a partir de múltiples fuentes de datos: HDFS, Apache Flume, Apache Kafka, Twitter y muchos más. Del mismo modo, Spark Streaming puede persistir los datos procesados en múltiples tipos de almacenamiento.



Figura 5. Documentación de Spark. Fuentes y destinos de datos en Spark Streaming.

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

Una vez construido un DStream a partir de una fuente de entrada de datos, se pueden aplicar sobre él dos tipos de operaciones:

Transformaciones

Las transformaciones generan un nuevo DStream.

Operaciones de salida

Son las operaciones de escritura en sistemas externos que nos permiten almacenar los resultados del procesamiento.

Las transformaciones aplicables a los DStreams son la mayoría de las que podemos usar con los RDDs (map, join, filter, etc.) —no olvidemos que un DStream es internamente una secuencia de RDDs—, además de nuevas operaciones relacionadas con el tiempo, como por ejemplo *ventanas* de procesamiento (window), que detallaremos más adelante.

Podemos desarrollar programas en Spark Streaming usando Java, Scala y Python. En este módulo usaremos Scala.

3.3. Un ejemplo simple

Para introducir el funcionamiento interno de Spark Streaming, trabajaremos con un ejemplo simple de procesamiento de un *stream*.

1

Para ello, usaremos la plataforma en la nube de Databricks. Desde <https://databricks.com/try-databricks> crearemos una cuenta gratuita en su Community Edition con la que podremos crear *notebooks* y experimentar con Spark Streaming de forma interactiva en un pequeño clúster en la nube y sin las complejidades de la instalación local.

The screenshot shows the Databricks website's landing page. At the top, there is a navigation bar with links for Blog, Resources, Partners, Documentation, Support, Careers, Contact Us, and a search icon. Below this is the Databricks logo and a secondary navigation bar with links for PRODUCT, APACHE SPARK, SOLUTIONS, CUSTOMERS, TRAINING, and EVENTS. A prominent blue button labeled 'TRY DATABRICKS' is located in the top right corner of the navigation bar. The main content area features the heading 'Select a version to get started.' followed by two columns: 'FULL-PLATFORM TRIAL' and 'COMMUNITY EDITION'. Each column lists features and has a 'START TODAY' button. A large red arrow points to the 'TRY DATABRICKS' button in the top navigation bar. In the bottom right corner, there is a blue button with a question mark icon and the text 'Ayuda'.

Blog Resources Partners Documentation Support Careers Contact Us Manage Account

databricks PRODUCT APACHE SPARK SOLUTIONS CUSTOMERS TRAINING EVENTS TRY DATABRICKS

Select a version to get started.

FULL-PLATFORM TRIAL
Put Apache Spark to work

- Unlimited clusters
- Notebooks, dashboards, production jobs, RESTful APIs
- Interactive guide to Spark and Databricks
- Deployed to your AWS VPC
- BI tools integration
- 14-day free trial (excludes AWS charges)

START TODAY

COMMUNITY EDITION
Learn Apache Spark

- Mini 6GB cluster
- Interactive notebooks and dashboards
- Public environment to share your work

START TODAY

Ayuda

Figura 6. Registro en Databricks Community para experimentar con Spark en cloud. Fuente: <https://databricks.com/try-databricks>

2

Una vez registrados, tendremos acceso al entorno de trabajo web.

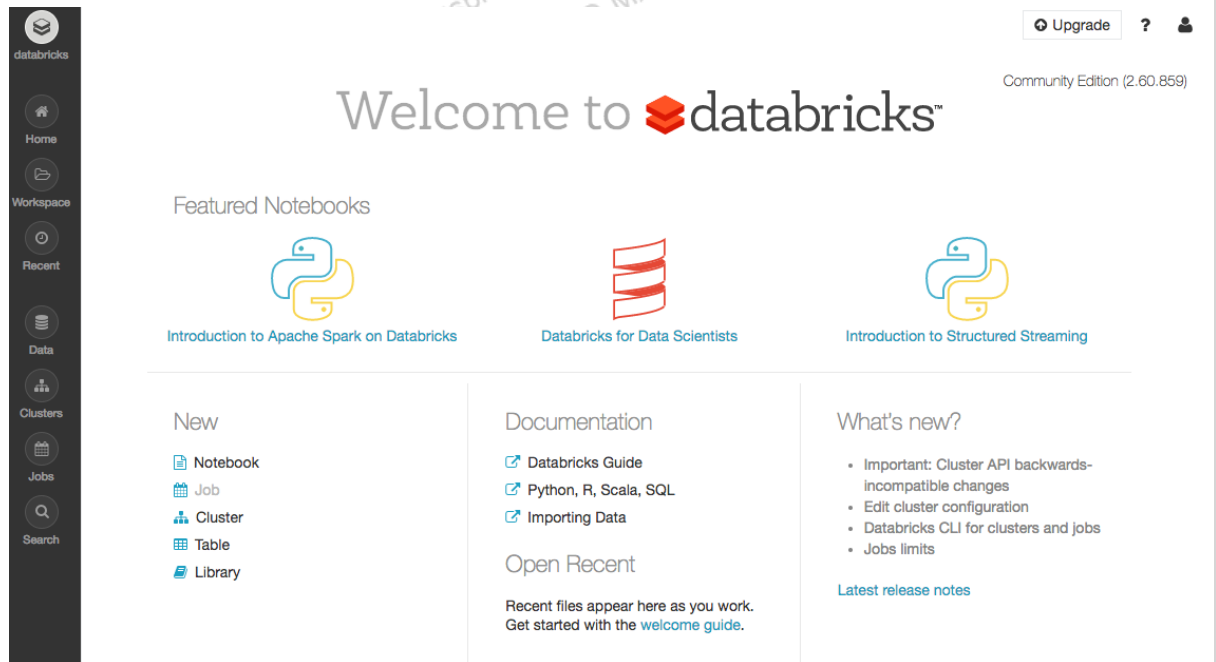


Figura 7. Captura de la pantalla principal del Databricks Workspace.

3

Comenzaremos por importar un *notebook* con un primer ejemplo sencillo. Para importar el *notebook* desde el Workspace procedemos como en la figura 8

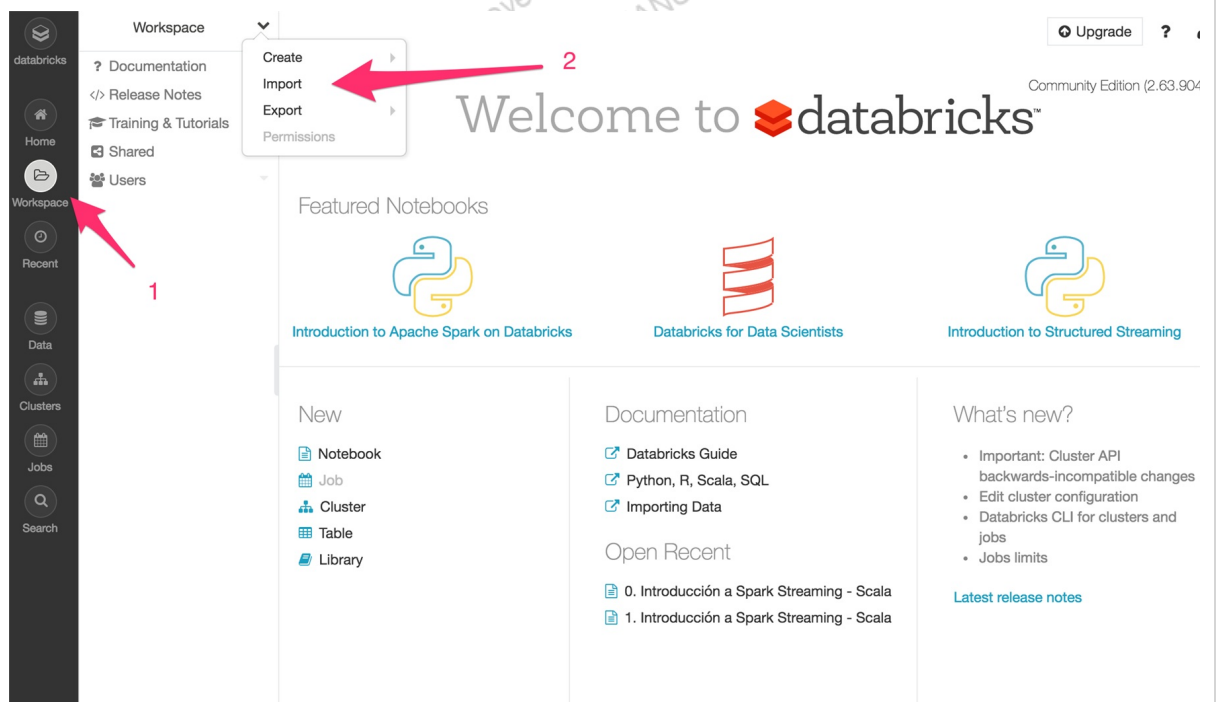
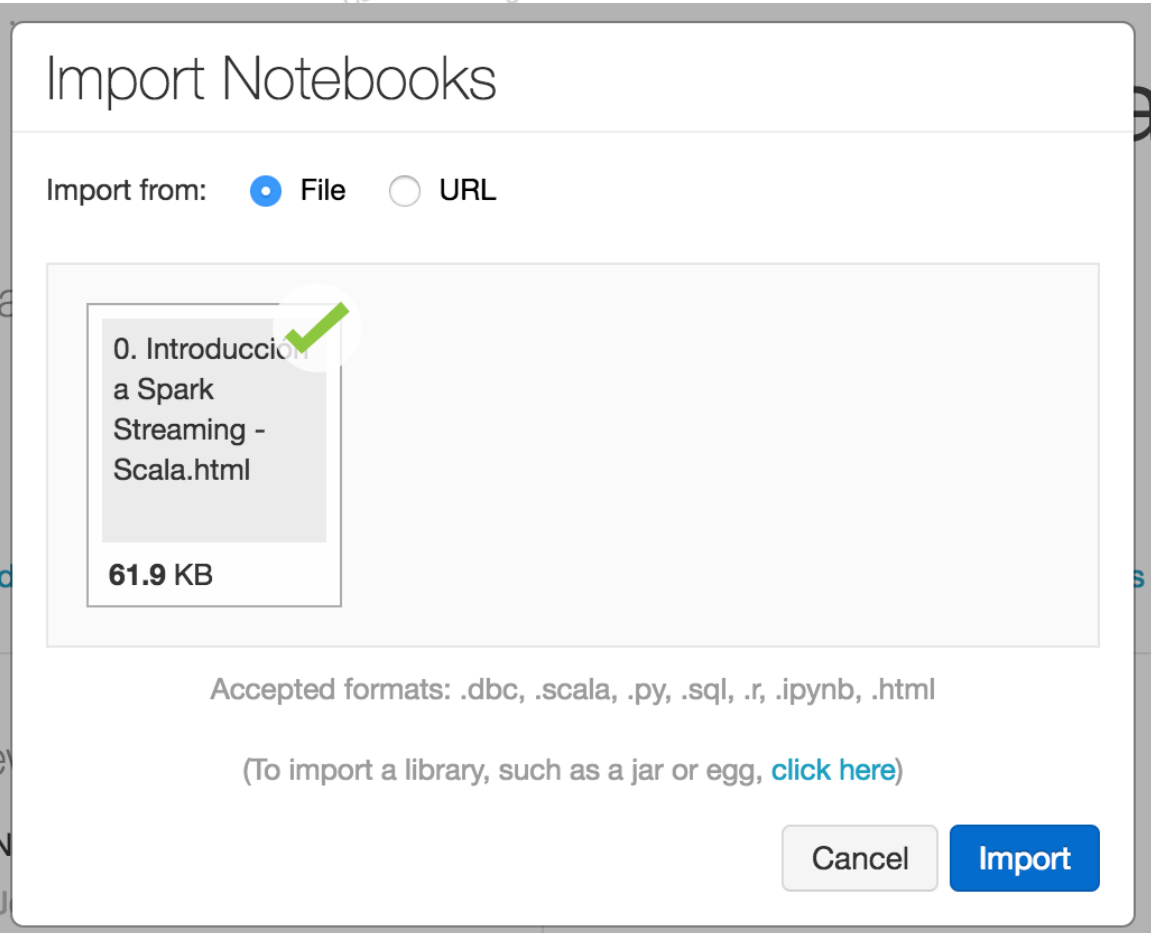


Figura 8. Importación de notebooks en Databricks Workspace.

Debemos importar el fichero 0. Introducción a Spark Streaming - Scala.html que se puede descargar en el siguiente enlace.

Figura 9. Importación de notebooks en Databricks Workspace.



Import Notebooks

Import from: ☒ File ☐ URL

0. Introducción a Spark Streaming - Scala.html
61.9 KB

Accepted formats: .dbc, .scala, .py, .sql, .r, .ipynb, .html

(To import a library, such as a jar or egg, [click here](#))

Cancel Import

Figura 10. Notebook suministrado e importado en Databricks Workspace.

0. Introducción a Spark Streaming - Scala (Scala)

Detached File View: Code Permissions Run All Clear Publish Comments Revision history

Cmd 1

Primer ejemplo de Streaming

En este ejemplo revisaremos los rudimentos básicos de Spark Streaming.

Para ello recibiremos un stream de líneas de texto del [registro de actividad de servidores web](#) en streaming. Por facilidad usaremos un generador en memoria de logs como fuente de datos en streaming.

Cmd 2

Configuración

En este sencillo ejemplo estableceremos el tamaño del microbatch.

Cmd 3

```
1 val batchIntervalSeconds = 1
```

Cmd 4

Importación de librerías

Para trabajar con streams tendremos que importar los paquetes adecuados.

Cmd 5

```
1 import org.apache.spark._
2 import org.apache.spark.streaming._
```

El *notebook* está comentado y pretende ser autoexplicativo. Para ejecutarlo, tenemos la opción de ejecutarlo entero ("Run all" en el menú) o celda a celda. En la primera ejecución, se arrancará un clúster para ejecutar el programa.

La anatomía de este ejemplo básico es simple y resume bien la estructura de un trabajo de procesamiento de *streams*:

Se crea un streaming context

Este contexto es el punto de entrada de toda la funcionalidad de streaming. El `StreamingContext` se crea con un tamaño de *microbatch* para especificar la frecuencia con la que se procesan los nuevos datos que llegan por el *stream*. En adelante, lo llamaremos *batch interval* y define el intervalo regular de tiempo en el que el `DStream` se va dividiendo en `RDDs`. En el *notebook* se define entonces que cada segundo se van a procesar los datos recibidos por el *stream*.

Se define el procesamiento que se ejecuta sobre cada microbatch o RDD

En nuestro ejemplo se filtran aquellas líneas que contienen un cierto texto usando la transformación `filter()` y a continuación se ejecuta una operación de salida `print()`. Es importante entender que en ese punto del código fuente solo se está definiendo el procesamiento, pero todavía no ha empezado.

Se inicia la recepción de datos

Esto es con `start()` sobre el contexto de *streaming*. Desde ese momento Spark Streaming comienza a crear *jobs* de Spark en el contexto subyacente [véase cómo se creó el contexto de *streaming*: `new StreamingContext(sc, Seconds(1))`, donde `sc` es el contexto de Spark Core]. Cada segundo se crea un nuevo *microbatch* y se procesa.

Se espera a la finalización de los trabajos

Como el procesamiento desde el comienzo de la recepción de datos se realiza en hilos separados, es necesario esperar a que finalicen usando `awaitTermination()` o alguna de sus variantes.

**Anotación: Notebook 1**

El [notebook 1. Introducción a Spark Streaming - Scala.html](#) (que se puede descargar en este enlace) contiene una versión mejorada del anterior. Básicamente crea el `StreamingContext` en una función e incluye el control de flujo necesario para reutilizar un contexto ya creado. Dicha estructura es la que se usará en los demás *notebooks*.

3.4. Arquitectura interna

Spark Streaming usa una arquitectura de *microbatches* en la que el procesamiento del *stream* se divide en una serie continua de procesamientos sobre pequeños *batches* de datos. Cada nuevo *batch* se genera en intervalos regulares de tiempo. Al comienzo de cada intervalo se crea un nuevo *batch* y los datos que llegan por el *stream* se añaden a dicho *batch* y al final del intervalo se deja de añadir datos. El tamaño del intervalo se define en unidades de tiempo y su valor queda a discreción del desarrollador (desde milisegundos a segundos).

1

Con cada *microbatch* se crea un RDD que se procesa con Spark para dar lugar a nuevos RDDs usando transformaciones. Los resultados de procesar cada *microbatch* se pueden, por ejemplo, almacenar en un sistema externo.

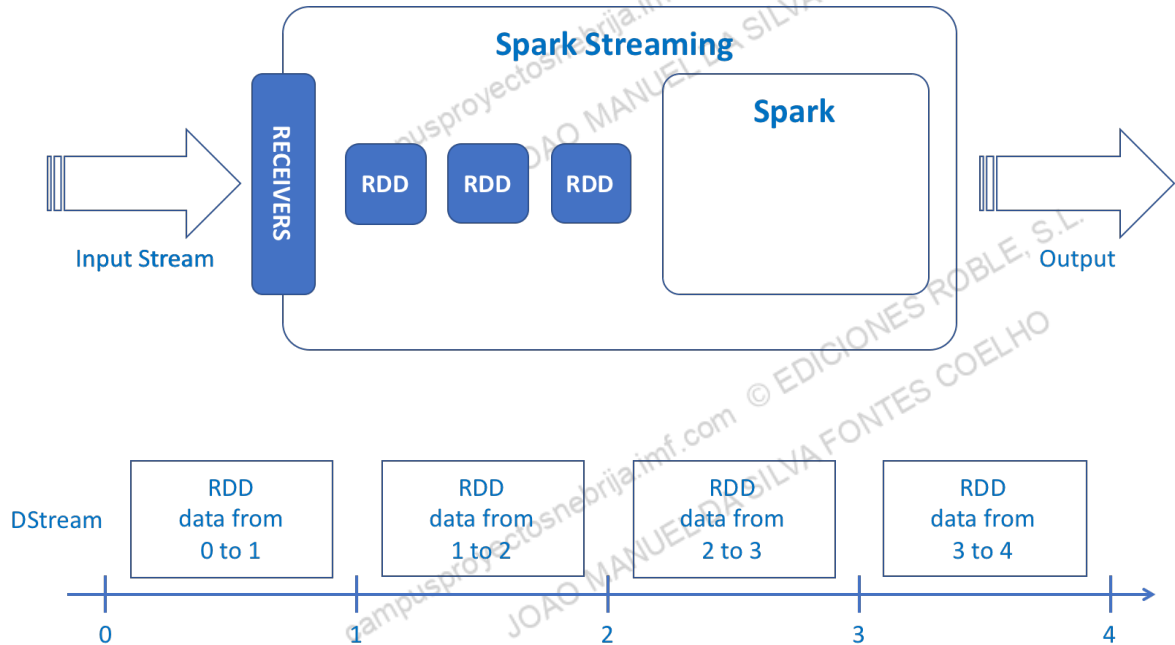


Figura 11. Discretized stream o DStream en Spark Streaming. Fuente: Elaboración propia.

Entonces, la abstracción que maneja Spark Streaming es el *discretized stream* o DStream, que no es más que una secuencia de RDDs donde cada uno contiene un trozo del *stream* de datos, como se observa en la figura 11.

Los DStreams se pueden crear desde fuentes de datos externas o realizando **transformaciones** sobre otros DStreams.

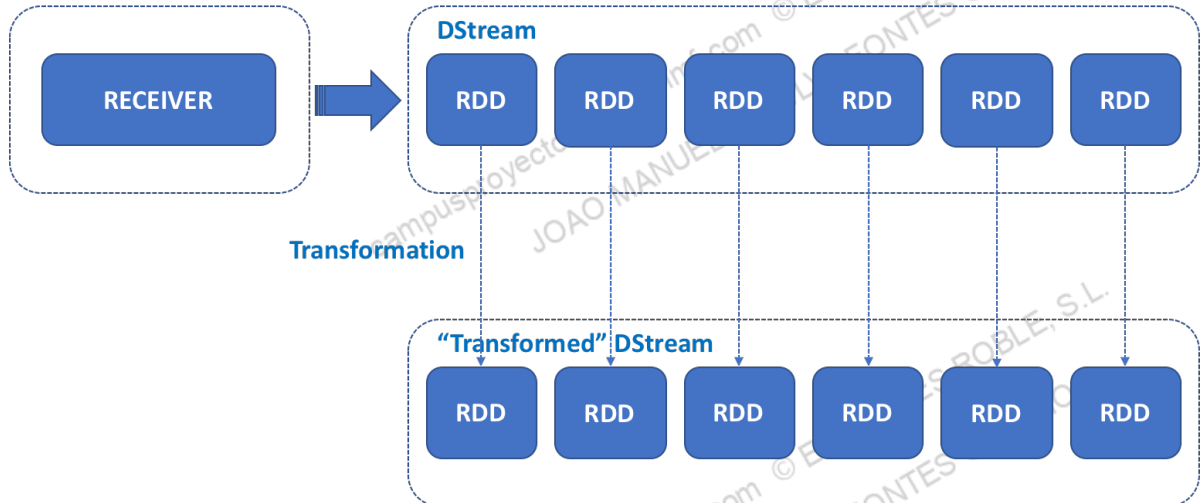


Figura 12. Transformación de un DStream en otro DStream. Fuente: Elaboración propia.

Además de transformaciones, sobre un DStream se pueden aplicar **operaciones de salida** hacia sistemas externos (como el `print()` de nuestro *notebook* que se realiza cada segundo). Estas operaciones son similares a las acciones que Spark permite hacer sobre RDDs, pero con la diferencia de que en Spark Streaming se ejecutan periódicamente en cada intervalo de tiempo y producen resultados en *batches*.

En la figura 13, se pueden observar los componentes de la arquitectura de ejecución de Spark Streaming.

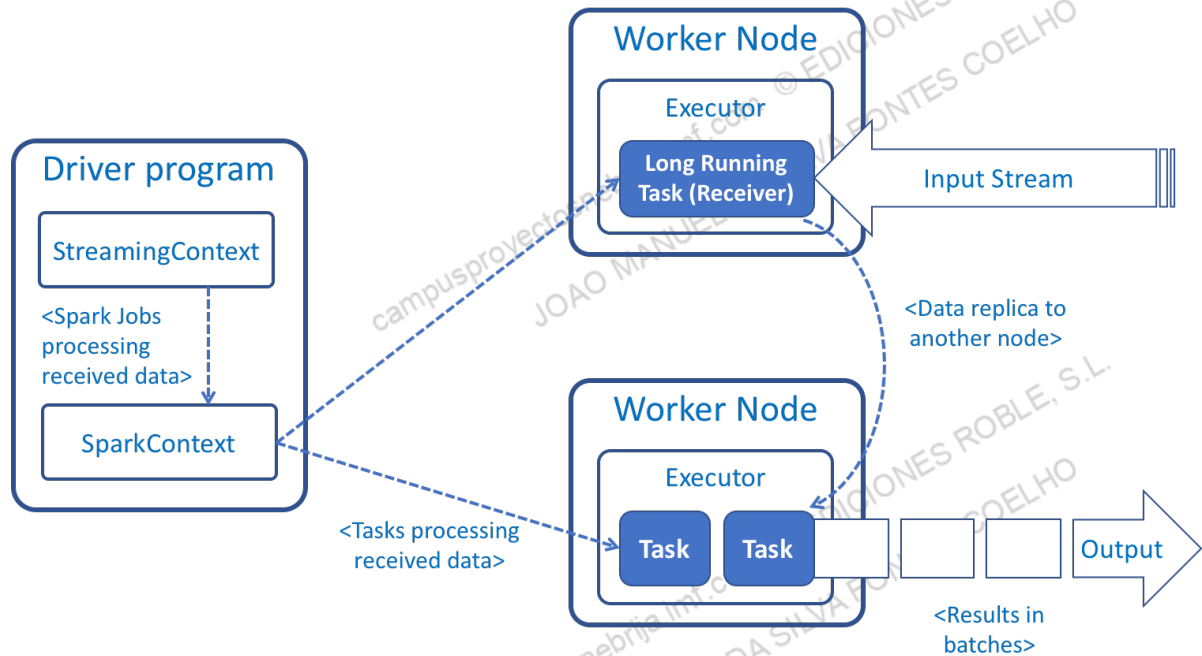


Figura 13. Arquitectura de ejecución de un programa Spark Streaming. Fuente: Elaboración propia.

Un programa de Spark Streaming (el *driver*) crea tantos *receivers* como fuentes de datos tenga. Esos *receivers* son procesos de larga ejecución que corren en procesos ejecutores en otros nodos del clúster de Spark. Los *receivers* se encargan de recolectar de manera continua los datos del *stream* y crear con ellos RDDs a intervalos regulares. Esos RDDs se replican en otros nodos del clúster y es allí donde se procesan en memoria.

El **StreamingContext** en el *driver* permite ejecutar periódicamente los *jobs* de Spark para procesar los datos y combinarlos con RDDs de momentos anteriores.

El procesamiento de *streams* es tolerante a fallos pues los datos recibidos se replican por defecto en dos nodos del clúster. Además, Spark Streaming guarda periódicamente el estado en un sistema de almacenamiento externo para facilitar la recuperación en caso de fallo (mecanismo de *checkpointing*).

3.5. Transformaciones

Spark Streaming permite dos tipos de transformaciones: sin estado (*stateless*) o con estado (*stateful*).

Las transformaciones *stateless* son aquellas en las que el procesamiento de un *microbatch* no depende de *microbatches* anteriores. Son transformaciones sin estado las que también se aplican a RDDs usando Spark Core: `map`, `filter`, `flatMap`, `groupByKey`, `reduceByKey`, etc.

Las transformaciones *stateful* usan resultados de *batches* previos para el procesamiento del *batch* en curso. En general son transformaciones basadas en ventanas de tiempo o información de estado que se mantiene a lo largo del tiempo.

3.5.1. Stateless transformations

Son las que, cuando se aplican, solo lo hacen sobre el RDD del *microbatch* actual. No se van a aplicar sobre todo el DStream (recordemos que el DStream se subdivide internamente en una secuencia de RDDs).

En la tabla 1 se muestran algunas de ellas:

Función	Propósito
<code>map(f)</code>	Aplica la función a cada elemento del DStream y devuelve un DStream con el resultado.
<code>filter(f)</code>	Devuelve un DStream con solo los elementos que pasan las condiciones de la función <code>f</code> .
<code>reduceByKey(f)</code>	Combina valores con la misma clave en cada <i>microbatch</i> .
<code>groupByKey(f)</code>	Agrupar valores con la misma clave en cada <i>microbatch</i> .
<code>union(otherStream)</code>	Devuelve otro DStream con la unión de elementos de ambos.
<code>join(otherStream)</code>	Usándolo sobre DStream de tipo (K, V) y (K, W) , devuelve un nuevo DStream de tipo $(K, (V, W))$ con todos los pares de elementos para cada clave.

Tabla 1. Algunas transformaciones de Spark Streaming.

Siguiendo con el ejemplo de nuestro *notebook*, si quisiéramos contar los *logs* por dirección IP, podríamos aplicar las transformaciones `map` y `reduceByKey` de este modo:

```
val parsedLogs = lines.map(_ => ApacheLogParser.parseLine(_))

// ApacheLogParser sería una clase de utilidad que

// extrae la información relevante de cada línea

val ip = parsedLogs.map(_ => (_.getIP(), 1))

val ipCount = ip.reduceByKey((x, y) => x + y)
```

Existe una transformación especial, `transform()`², que permite operar directamente sobre los RDDs del DStream. Esta transformación permite aplicar cualquier función que transforme RDDs en RDDs y su utilidad principal es la reutilización de código desarrollado para procesamientos *batch* para usarla en procesamientos en *streaming*.

```
val nuevoDStream = lines.transform { rdd => funcionAREutilizar(rdd) }
```

En la documentación³ de Spark, se puede encontrar una lista más exhaustiva de estas transformaciones.

²Apache Spark 2.3.0. Spark Streaming Programming Guide: "Transform Operation". [En línea] URL disponible en: <https://spark.apache.org/docs/latest/streaming-programming-guide.html#transform-operation>

³Apache Spark 2.2.0. Spark Programming Guide: "Transformations" [En línea] URL disponible en: <https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#transformations>

3.5.2. Stateful transformations

1

Son, como hemos dicho, aquellas transformaciones que dependen de datos de anteriores *microbatches* para calcular resultados.

Hay dos tipos de transformaciones con estado:

1. Transformaciones sobre ventanas deslizantes⁴. Son las transformaciones `window()`, `countByWindow()`, `reduceByWindow()` y similares.
2. `updateStateByKey()`⁵ para mantener el estado de ciertos eventos a lo largo del tiempo.

⁴Apache Spark 2.3.0. Spark Streaming Programming Guide: "Window Operations". [En línea] URL disponible en: <https://spark.apache.org/docs/latest/streaming-programming-guide.html#window-operations>

⁵Apache Spark 2.3.0. Spark Streaming Programming Guide: "UpdateStateByKey Operation". [En línea] URL disponible en: <https://spark.apache.org/docs/latest/streaming-programming-guide.html#updatestatebykey-operation>

2

Las transformaciones con estado requieren habilitar *checkpointing* para que Spark Streaming pueda almacenar estados intermedios en almacenamiento persistente por motivos de tolerancia a fallos. Para hacerlo, es necesario invocar la función `ssc.checkpoint(...)` del contexto de *streaming*.

Las transformaciones con ventanas deslizantes permiten calcular resultados sobre un periodo de tiempo superior al intervalo configurado en el `StreamingContext` (el *batch interval*) y podemos así combinar resultados de múltiples *microbatches*. Estas transformaciones toman dos parámetros: la duración de la ventana y la duración del deslizamiento. La duración de la ventana es el número de *batches* anteriores que se tienen que considerar. La duración del deslizamiento controla con qué frecuencia se calculan resultados sobre el `DStream` (por defecto cada *batch interval*).

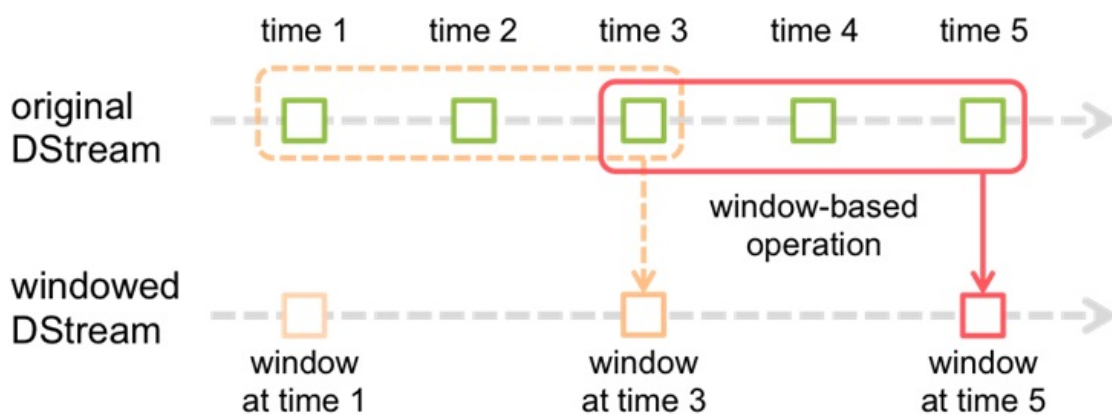


Figura 14. Documentación de Spark. Transformación de tipo ventana. Fuente: <https://spark.apache.org/docs/latest/streaming-programming-guide.html#window-operations>

3

En la figura 14 se muestra un DStream al que se le aplica una transformación en ventana para dar lugar a un nuevo DStream. La duración de la ventana es equivalente a 3 *batch interval*, es decir, se consideran tres *batches* anteriores. El deslizamiento es 2 porque cada dos *microbatches* se realizan los cálculos que dan lugar a *microbatches* en el nuevo DStream.

Por ejemplo, si quisiéramos contar cada 10 segundos el número de *logs* que se producen durante el minuto anterior:

```
val logWindow = lines.window(Seconds(60), Seconds(10))
```

```
val counts = logWindow.count()
```

La transformación `window()` es la más sencilla, pero disponemos de transformaciones más especializadas y eficientes como `countByWindow`, `reduceByWindow`, `reduceByKeyAndWindow` y `countByValueAndWindow`.

En muchas ocasiones es útil mantener el estado entre *batches* del DStream. La transformación `updateStateByKey()` nos sirve para mantener una variable de estado durante todo el *stream*. Esta transformación toma como entrada un DStream de pares (clave, evento) y devuelve un nuevo DStream de pares (clave, estado). Para ello es necesario darle como argumento la función que permite actualizar el estado de cada clave cada vez con cada evento.



Anotación: Notebook 3

El *notebook* 3, que se puede descargar en el siguiente enlace: [3. Introducción a Spark Streaming - Scala.html](#), muestra un ejemplo en el que se crea un *stream* de pares clave-valor (código_http, 1) cada vez que se lee una línea de *logs* en el *stream*. Se define además una función `updateRunningSum` que es la que se usará para incrementar el número de ocurrencias de cada código http. Se usa entonces la transformación con estado `updateStateByKey(updateRunningSum _)` para generar un nuevo *stream* de pares clave-valor (código_http, contador_de_veces). Si se ejecuta el *notebook*, se verá que el contador de veces que se lee un código http se va incrementando a lo largo del tiempo según procesamos el *stream* (no es un cálculo solo para el *microbatch* en curso). En el ejemplo, se configura además el *checkpointing* sobre *filesystem* para que Spark Streaming pueda guardar el estado de acumulados en caso de que se necesite recuperar un nodo de procesamiento.

3.6. Operaciones de salida

Las operaciones de salida especifican qué se hará con los datos transformados en el *stream*.

En los *notebooks* de ejemplo estamos volcando en salida estándar los resultados de las transformaciones en el DStream usando la operación `print()`. Esta operación nos resultará útil para trazar nuestros programas durante el desarrollo.

Otra operación de salida es `save()` y permite almacenar en disco los resultados de la transformación en cada *microbatch*.

Por ejemplo: `statusCodesCount.saveAsTextFiles("directory", ".txt")` guardará los resultados de cada *batch* en subdirectorios de `directory` en ficheros con extensión `.txt` a la finalización de cada *microbatch*.

Del mismo modo, existen operaciones de salida para almacenar los resultados en bases de datos relacionales o NoSQL.

3.7. Integración con fuentes de datos

Spark Streaming soporta diferentes tipos de fuentes de datos en *streaming*.

El Streaming Context soporta de manera nativa fuentes de tipo *filesystem* y conexiones *socket*.

Usando clases de utilidad es posible conectar con fuentes más avanzadas como *brokers* de Kafka, Flume, Kinesis, actores Akka, etc.



Anotación: Algunas fuentes de datos en streaming

Tabla 2. Algunas fuentes de datos en streaming para Spark Streaming.

Fuente	Documentación
Apache Kafka 0.8.x	https://spark.apache.org/docs/latest/streaming-kafka-0-8-integration.html
Apache Kafka 0.10.x	https://spark.apache.org/docs/latest/streaming-kafka-0-10-integration.html
Apache Flume	https://spark.apache.org/docs/latest/streaming-flume-integration.html

En todos los casos la documentación muestra los artefactos que es necesario incluir como dependencias⁶ de nuestro programa y el código para configuración de la fuente (direcciones IP, puertos, tópicos, etc.) para cada tipo de fuente.

Es importante tener en cuenta que los receptores de datos son procesos de larga duración que se ejecutan en los nodos *worker* del clúster y ocuparán cada uno un *core* de procesador. Si un *worker node* tiene 4 *cores* a su disposición y usamos 4 receptores de datos en *streaming*, no quedarán entonces *cores* disponibles para procesar los datos. Este tipo de consideraciones son importantes a la hora de dimensionar un clúster de Spark para trabajos en *streaming*.

⁶Apache Spark 2.3.0. Spark Streaming Programming Guide: "Linking". [En línea] URL disponible en: <https://spark.apache.org/docs/latest/streaming-programming-guide.html#linking>

3.8. Garantías de procesamiento

1

Una de las principales ventajas de Spark Streaming es la garantía que ofrece para tolerar fallos.

En tanto en cuanto los datos de entrada del *stream* estén almacenados de manera fiable (por ejemplo, en un tópico de Kafka, en *filesystem* distribuido, etc.) Spark Streaming garantiza semántica de procesamiento *exactly once*. Es decir, que cada mensaje se procesará una única vez⁷ incluso en caso de fallos en el clúster (caídas de nodos o fallos de red).

El primer paso para que nuestros programas de Spark Streaming se beneficien de estas garantías de procesamiento es habilitar el *checkpointing* (en el *notebook* 3. Introducción a Spark Streaming - Scala.html fue necesario habilitar el *checkpointing* para usar transformaciones con estado). Se trata del mecanismo principal para garantizar la tolerancia a fallos. Spark Streaming volcará periódicamente los datos que se manejan en el procesamiento en almacenamiento persistente (HDFS o disco local, por ejemplo) y de ahí los leerá en caso de necesitar recuperarse frente a un fallo del sistema.

⁷Li Zhang's Blog: "How to achieve exactly-once semantics in Spark Streaming". [En línea] URL disponible en: <http://shzhangji.com/blog/2017/07/31/how-to-achieve-exactly-once-semantics-in-spark-streaming/>

2

Spark Streaming usa *checkpointing* por dos motivos. Spark es capaz de recalcular el estado a partir del grafo de transformaciones, pero en Spark Streaming es necesario limitar cuánto se va hacia atrás en el tiempo (hasta el último *checkpoint* almacenado) pues un *stream* puede ser infinito. Por otra parte, los *checkpoints* permiten recuperar el estado del *driver* (el programa "cliente" que controla el procesamiento) en caso de caída del proceso.

La tolerancia a fallos en los *worker nodes* es la misma en Spark Streaming que en Spark. Todos los datos que se reciben de fuentes externas se replican entre *workers*. Todos los RDDs creados mediante transformaciones de esas fuentes son tolerantes a fallos de nodos *worker* pues Spark es capaz de reconstruir un RDD a partir de su linaje de transformaciones desde los datos de entrada replicados. Todas las transformaciones que se hacen en Spark Streaming tienen garantías *exactly-once*.

3

En caso de fallo de un receptor en Spark Streaming, un nuevo receptor se levantará en otro nodo *worker* del clúster, pero la tolerancia a fallos depende de la naturaleza del sistema origen: si este es capaz de reenviar o no los datos. Si usamos HDFS o consumición de tópicos de Kafka, no existe problema pues la fuente se considera confiable en tanto en cuanto podemos leer de nuevo los datos desde donde ocurrió el fallo (son ejemplos de recepción *pull based*, para los que Spark Streaming se ocupa de “traer” los datos), pero en el caso de Twitter o Flume usado en modo *push* (en general en cualquier tipo de fuente de tipo *push*, que son aquellas que “mandan” los datos a Spark) es posible perder datos.

Para las operaciones de salida es necesario tener presente que, en caso de fallo, la operación puede ejecutarse más de una vez. Tienen semántica *at-least-once*. Por ello, es responsabilidad del desarrollador implementar operaciones *idempotentes*⁸.

⁸*Ibidem.*

3.9. Consideraciones de rendimiento

Más allá de las consideraciones que se aplican a Spark en general, para trabajos de Spark Streaming hay algunas cuestiones específicas que se deben tener en cuenta:

Tamaño de la ventana de procesamiento

El **tamaño de la ventana de procesamiento** (el *batch interval*). Se recomienda ajustar este parámetro empezando en el rango de segundos y disminuirlo progresivamente hasta el rango de los 500 milisegundos. En el momento en el que el rendimiento deja de ser consistente es cuando habremos identificado el *batch interval* más apropiado para nuestro trabajo de *streaming*. Para el caso del procesamiento con operaciones en ventana, cabe poner especial atención a la hora de seleccionar el *slide interval* para que no constituya un cuello de botella.

Grado de paralelismo

Una de las formas más directas de reducir el *batch interval* es incrementar el paralelismo de procesamiento. Para ello es posible actuar de varias formas: incrementar el número de receptores, reparticionar explícitamente los datos recibidos o incrementar el paralelismo de las transformaciones de agregación.

IV. Resumen



En esta unidad nos hemos detenido a revisar y experimentar las capacidades de Spark para procesar *streams* de datos. Spark Streaming usa el concepto de DStreams para dividir el procesamiento del *stream* en *microbatches*. De este modo, todos los conceptos aprendidos con procesamiento *batch* usando Spark nos sirven para el procesamiento de *streams* de datos. Spark Streaming introduce transformaciones específicas para *streaming* (ventanas de procesamiento). Spark Streaming hereda las garantías de procesamiento de Spark.

Ejercicios

Caso práctico



Importa el [notebook: 2. Introducción a Spark Streaming - Scala.html](#) (que se puede descargar en este enlace) en el Workspace de Databricks. En el *notebook* está resuelto el cálculo del tamaño total en bytes servido por el Apache en cada *microbatch*.

Ejercicio 1

Modifica el *notebook* para filtrar los *logs* de una IP concreta, por ejemplo, la 188.45.108.168.



Anotación: Solución

Se podría hacer con una transformación *stateless* simple sobre el DStream:
`accessLogs.filter(_clientIp.equals("188.45.108.168"))`

A continuación, puedes descargar el notebook con la [solución](#).

Ejercicio 2

Modifica el *notebook* para contar el número de líneas de *log* que se procesan por *microbatch*.



Anotación: Solución

Bastaría, por ejemplo, con una transformación *stateless* como `contentSizes.count.print()`

A continuación, puedes descargar el notebook con la [solución](#).

Ejercicio 3

Modifica el resultado del ejercicio anterior para hacer el cálculo en ventana de modo que se calcule cada dos segundos el total de líneas de *log* en los 5 segundos anteriores.



Anotación: Solución

Se crea un nuevo DStream usando window(Seconds(5), Seconds(2)) y se hace el cálculo del total (la transformación count) sobre él.

A continuación, puedes descargar el notebook con la [solución](#).

Recursos

Enlaces de Interés



Blog de Databricks: “Structured Streaming In Apache Spark. A new high-level API for streaming”

<https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html%20>



Documentación oficial de Apache Spark 2.3.0: “Spark Streaming Programming Guide”.

<https://spark.apache.org/docs/latest/streaming-programming-guide.html>



Documentación oficial de Apache Spark 2.3.0: “Structured Streaming Programming Guide”.

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

Glosario.

- **Batch interval:** Duración temporal del microbatch en el que Spark Streaming divide el stream.
- **DStream (discretized stream):** Abstracción de Spark Streaming que representa un flujo continuo de datos como una secuencia de RDDs.
- **Microbatch:** Un RDD de entre los que se divide el DStream.
- **RDD (Resilient Distributed Dataset):** Abstracción básica de datos en Spark que representa una colección inmutable y particionada de elementos sobre los que Spark realiza procesamiento en paralelo.
- **Transformación:** Operación básica en Spark para transformar datos. En realidad, transforma un DStream (o un RDD) en otro DStream (u otro RDD).