

# Spark Streaming

Sistemas Distribuidos de Procesamiento de Datos

Autores: Felipe Ortega

Máster en Data Science. ETSII.

Móstoles, Madrid

14 de mayo de 2020



Universidad  
Rey Juan Carlos

# Contenidos

- 1 Arquitecturas procesamiento *streaming*
- 2 DStreams
- 3 Transformaciones en ventana (*windowed*)

# Introducción Spark Streaming

- Se trata de un módulo que permite trabajar con **flujos de datos** que recibimos en **tiempo real**, de forma que los resultados de las operaciones aplicadas sobre los datos (transformaciones) se vayan actualizando automáticamente.
- Es posible configurar el **intervalo de actualización**, hasta un mínimo de 0.5 segundos (suficiente para un gran número de aplicaciones).
- El concepto fundamental sobre el que gira el funcionamiento de Spark Streaming es el de **discretized streams** o **DStreams**.
- También proporciona cierto nivel de **tolerancia a fallos**, así como integración con el resto de componentes de Spark (RDDs, etc.).

## Section 1

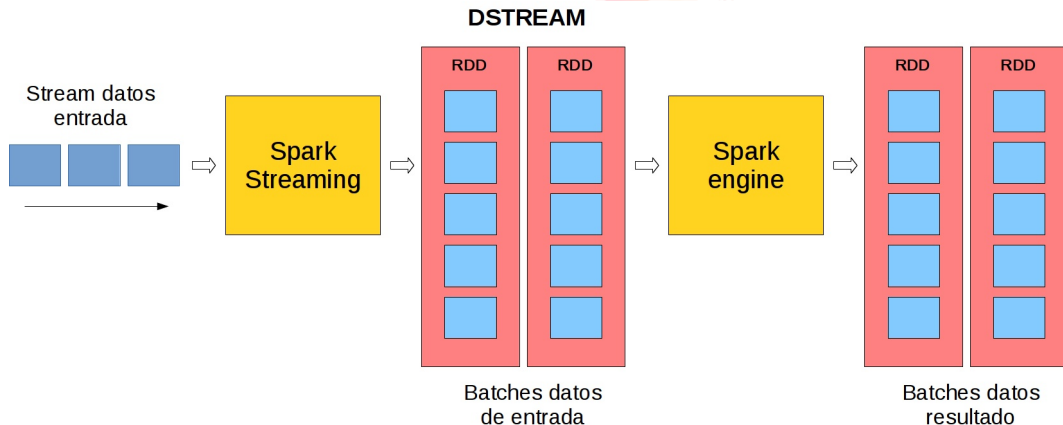
# Arquitecturas procesamiento *streaming*

# Arquitectura Spark Streaming



**Figura:** Diagrama conceptual arquitectura Spark Streaming.

# Arquitectura Spark Streaming



**Figura:** Diagrama detallado arquitectura Spark Streaming.

# Arquitecturas híbridas de procesamiento de datos

Uno de los principales alicientes de Spark es poder combinar en la misma solución capacidades de procesamiento de datos en modo *batch* y modo *streaming*. En el ámbito de *ciencia de datos*, esta propuesta se conoce como **arquitectura híbrida**.

- En la actualidad, existen dos propuestas al respecto de cómo combinar procesamiento *batch* y *streaming* en un mismo sistema:
- **Arquitectura lambda** [1]: Dos niveles de procesado de datos: *capa speed* (streaming) y *capa batch*, con una interfaz de acceso al servicio común.
- **Arquitectura kappa** [2]: Solo tenemos capa streaming, he intentamos satisfacer las consultas recomputando sobre un conjunto base de datos en un intervalo predefinido de antemano.

# Arq. *lambda* vs. *kappa*

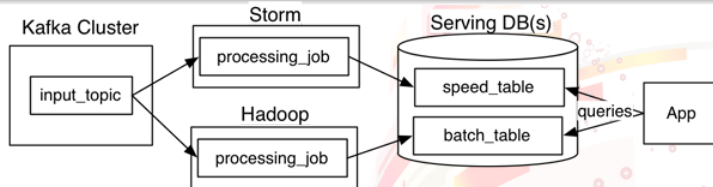


Figura: Arq. *lambda*

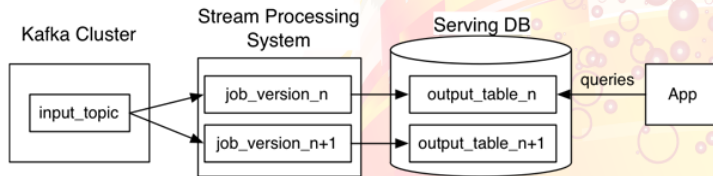


Figura: Arq. *kappa*



# Arquitectura lambda

Dos niveles de procesado:

- **Capa batch:** Se encarga del procesamiento periódico del volumen masivo de datos adquiridos hasta el momento por el sistema (descriptiva global, métricas totales, etc. Normalmente ejecutado cada día (o en periodos más largos).
- **Capa speed:** Proporciona resultados actualizados en tiempo real basados en los nuevos datos adquiridos más recientemente por el sistema. Es decir, completa lo que ha ocurrido desde la última ejecución de la capa batch hasta el momento actual.

# Arquitectura lambda

## Ventajas:

- **Secuencia de elementos inmutables:** Esta arquitectura pone énfasis en modelar las transformaciones aplicadas a los datos como operaciones que dan como resultado nuevos conjuntos inmutables de datos (una vez aplicada no se puede volver atrás). Spark opera por defecto de esta forma (incluso con RDDs no streaming).
- **Re-procesado de datos:** Resalta el hecho de que si hacemos cambios en el código o algoritmos, necesariamente tenemos que volver a efectuar los cálculos sobre todos los datos (al menos, hasta el máximo intervalo histórico que hemos sido capaces de almacenar).

# Arquitectura lambda

Mito:

- **Es capaz de solventar el problema del teorema CAP:** No es cierto.
- Pueden aparecer discordancias entre los datos adquiridos y los resultados presentados al usuario.
- Y no, ningún otro sistema actual, por el momento, es capaz de resolver estos problemas, ni siquiera aquellos específicamente diseñados para procesado (en verdad) de datos streaming.

<http://ferd.ca/beating-the-cap-theorem-checklist.html>

# Arquitectura lambda

## Inconvenientes:

- El principal es que obliga a duplicar el código/algoritmos en las dos capas de la arquitectura.
- Por tanto, fuerza a mantener una buena disciplina aplicando todos los cambios que se hagan en una capa a la otra y viceversa → **mantenibilidad**.
- Esto es mucho peor (con diferencia) que la situación original, ya que es difícil mantener la consistencia de ambos sistemas.

# Arquitectura kappa

- Propuesta: Usar *batch* cuando no tenemos requisitos temporales estrictos, y *streaming* (solo) cuando sí que los tenemos.
- Si tenemos que volver a procesar los datos (desde el último punto de recuperación disponible), volver a lanzar otro trabajo *streaming* y cuando éste alcance al anterior redirigir la salida para que apunte al nuevo trabajo y borrar el original.
- Variantes de esta aproximación se encuentran formalmente descritas (e implementadas) en algunos sistemas de procesamiento *streaming* como Apache Samza (como Kafka, creado en LinkedIn), tal y como se ha visto en la asignatura previa SDPD 2.

# La aproximación de Spark

- Spark intenta asumir el rol de “navaja multiusos” en un solo producto (distintas interfaces para un mismo motor de procesamiento).
- Con otras opciones, necesitamos **sistemas diferentes** para implementar cada estrategia (e.g. Hadoop para *batch*, Storm, Samza o Flink para *streaming*).
- Cada opción en su área puede ofrecer un muy buen rendimiento, pero debemos sopesar la ventaja de instalar, administrar y utilizar un único sistema que pueda atacar estos problemas en un estilo *best effort*.
- En este sentido, dependiendo de nuestro diseño Spark se puede adaptar tanto a una arquitectura *lambda* como a una *kappa*, pero usando un único producto.
- Además, tenemos la ventaja de poder usar otras APIs de Spark (DataFrames y consultas SQL) sobre DStreams.

## DStreams

## Section 2

## DStreams

# Definición de DStream

- Reunimos paquetes de datos de los flujos de entrada, capturados cada cierto intervalo de tiempo, que se presentan como un RDD.
- Por tanto, los datos de entrada de Spark Streaming son una secuencia continua de RDDs (recordemos, inmutables).
- Esta secuencia de RDDs de entrada se denomina **discretized stream** o **DStream**.
- Los datos se procesarán mediante un grafo de **transformaciones** (al igual que con RDDs ordinarias), cada una de las cuales dará como resultado otra RDD (inmutable).
- Como veremos, las transformaciones pueden ser genéricas o bien (ya predefinidas en la biblioteca de Spark Streaming) o podemos crear nuestras propias transformaciones más complejas.



# Adquisición de datos: receptores

- Los **receptores** son objetos que modelan orígenes de datos *streaming* para este módulo de Spark. Existen diferentes tipos:
  - **Fuentes básicas:** disponibles directamente en la biblioteca de Spark Streaming, tales como ficheros/directorios, HDFS, serialización Avro y Parquet, sockets o actores **Akka**.
  - **Fuentes avanzadas:** **Kafka**, **Flume**, **Kinesis (AWS)**, Twitter, etc. Precisan que importemos explícitamente algunas bibliotecas o componentes adicionales.
- **Recursos de ejecución:** Para dimensionar recursos, es conveniente recordar que cada receptor usará un hilo ejecución y un core de CPU asociado para su ejecución **en exclusiva**. Por tanto, para procesar  $N$  fuentes en paralelo necesitamos al menos  $N + 1$  cores.

# Programación con DStreams

- El punto de acceso a la API de Spark Streaming es la clase `pyspark.streaming.StreamingContext`.
- A continuación usaremos el objeto `StreamingContext` para acceder al resto de funciones de la API.

```
from pyspark.streaming import StreamingContext
# Suponemos que 'sc' es un objeto de tipo SparkContext
# creado a partir de un SparkSession
ssc = StreamingContext(sc, 3) # Arg. adic. frecuencia de refresco
```

# Programación con DStreams

- El siguiente programa crea un flujo de procesamiento streaming que cuenta las palabras recibidas por el puerto 10001 de la máquina local.

```
lines = ssc.socketTextStream("localhost", 10001)
words = lines.flatMap(lambda line: line.split(""))
weights = words.map(lambda word: (word, 1))
counts = weights.reduceByKey(lambda x, y: x + y)
# Muestra los primeros resultados por pantalla
counts.pprint()
# Lanza el sistema de procesamiento streaming
ssc.start()
```

# Programación con DStreams

**Ejercicio 1:** Ejecutar en la máquina local con Spark el programa anterior (disponible en `networdcount.py`).

- 1. Abrir una terminal y ejecutar el comando `nc` en Linux o Mac (`ncat` en Windows, que forma parte de `nmap`).

```
# Desde una terminal  
nc -lk 10001
```

- 2. Enviar a Spark local para su ejecución el archivo `networdcount.py` de la siguiente forma:

```
# Desde otra terminal, en el directorio $SPARK_HOME  
./bin/spark-submit networdcount.py
```

# Acerca de Streaming Context

- El DAG de operaciones no se evalúa hasta que se ejecuta el `StreamingContext` con `start()` (igual que en RDDs no se evalúa hasta llamar a una acción).
- No es posible alterar el DAG de un `StreamingContext` una vez iniciado con `start()`.
- No podemos volver a lanzar un `StreamingContext` una vez que se ha parado con `stop()`.
- En cada JVM solo podemos tener un `StreamingContext` ejecutándose simultáneamente.
- Cuando se llama a `stop()` sobre un `StreamingContext` también se detiene el `SparkContext` asociado (comportamiento configurable con un parámetro de `stop()`).
- Un mismo objeto `SparkContext` puede usarse para definir múltiples `StreamingContext`, siempre que se pare cada uno de ellos antes de crear el siguiente (y que no se detenga el `SparkContext` asociado).

# Programación con DStreams

Transformaciones **básicas** sobre un DStream:

- `count()`.\*.
- `countByValue()`.\*.
- `flatMap(funcion)`.
- `filter(condicion)`.
- `map(funcion)`.
- `reduce(funcion)`.
- `reduceByKey(funcion, [num_tareas])`.
- `repartition(num_particiones)`.

(\*) Operaciones que eran *acciones* aplicadas sobre un RDD pero son *transformaciones* aplicadas sobre un DStream.

# Programación con DStreams

- `count()`: Retorna un DStream de un solo elemento con el conteo del número de elementos contenidos en cada RDD del DStream de entrada.
- `countByValue()`: Si se aplica sobre un DStream con elementos tipo clave-valor (K,V), devuelve un DStream con elementos (K, Long) que cuenta la frecuencia de aparición de la clave en cada RDD del DStream de entrada.
- `flatMap(funcion)`: Se aplica sobre cada elemento de un DStream para devolver otro DStream cuyo contenido son iteradores para recorrer los valores resultantes.
- `filter(condicion)`: Como en RDDs, elimina del DStream de entrada los elementos que cumplen la condición indicada.
- `reduce(funcion)`: Retorna un DStream con una RDD de un solo elemento, resultado de agregar todos los elementos de cada RDD del DStream de entrada según la operación que se indica como argumento (debe ser asociativa para poder paralelizar).

# Arquitectura Spark Streaming

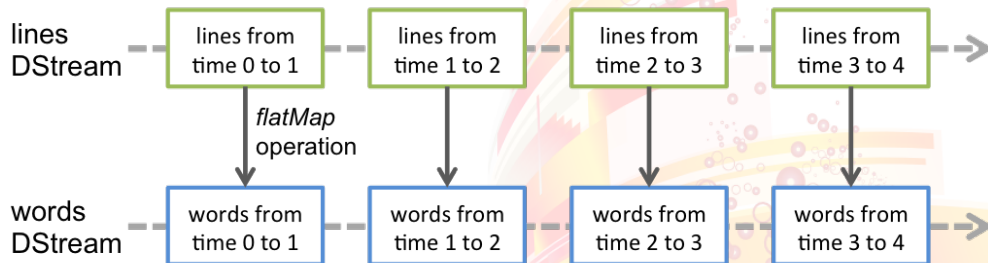


Figura: Operaciones sobre DStreams.



# Programación con DStreams

## Transformaciones **avanzadas**:

- `union(DStream)`: Sustituye los valores faltantes por un nuevo valor, que pasamos como argumento. Podemos indicar sobre qué columnas queremos operar.
- `join(DStream, [num_tareas])`: Une dos DStreams de pares  $(K,V)$  y  $(K,W)$  *rightarrow*  $(K, (V,W))$ , para todos los pares de valores de cada clave.
- `cogroup(DStream, [num_tareas])`: Similar al anterior, pero devuelve un DStream en el que los valores son tuplas  $(K, \text{collection}[V], \text{collection}[W])$ .
- `updateStateByKey(funcion)`: El DStream que devuelve retiene información de estado, y actualiza el valor de cada clave según la función indicada, el valor anterior y el nuevo valor para esa clave. Los datos de estado de cada clave se pueden definir libremente.
- `transform(funcion)`: Permite definir libremente la función que se aplica a cada RDD del DStream de entrada para dar como resultado el DStream de salida.

# Programación con DStreams

**Ejercicio 2:** ejemplo de función de actualización de estado según la clave de cada elemento `updateStateByKey(...)`.

- Archivo código Spark Streaming: `networdcount_state.py`.
- Las instrucciones de ejecución son las mismas que en el caso anterior, pero la salida es diferente.

# Programación con DStreams

## Operaciones de salida:

- `pprint()`: En modo driver, muestra por pantalla los 10 primeros elementos de cada lote de un DStream.
- `saveAsTextFiles(prefijo, [sufijo])`: Guarda el DStream de resultado como ficheros de texto, con la nomenclatura `prefijo-timestamp.sufijo`.
- `foreachRDD(funcion)`: Aplica la función pasada como argumento a cada RDD que genera el DStream. El cálculo se hace en el proceso driver que ejecuta la aplicación, después de que se hayan aplicado el DAG completo de transformaciones.

Transf. en ventana

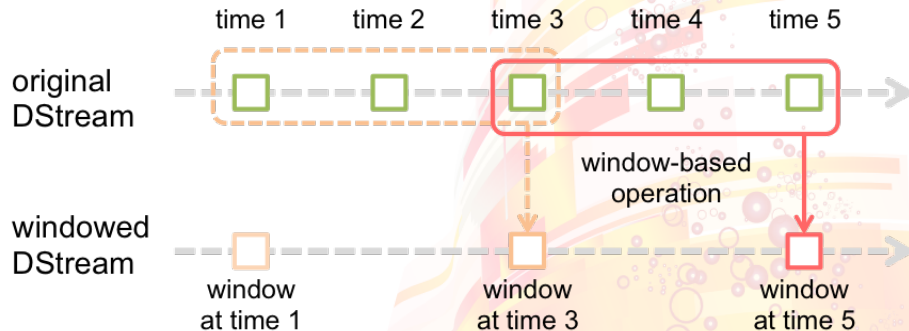
## Section 3

Transformaciones en ventana (*windowed*)

# Programación con DStreams

- Lo más frecuente es que analicemos los datos del flujo conforme se van recibiendo. Por tanto, no se analiza el flujo de datos en su totalidad.
- En todo caso, podemos mantener como hemos visto información de estado para ciertas claves que se va actualizando conforme llegan nuevos datos.
- Sin embargo, otro caso de uso común consiste en efectuar cálculos sobre los datos recibidos durante cierto intervalo de tiempo (los últimos 10 minutos o la última hora).
- Este es el concepto de **operaciones sobre ventana de datos**, que permiten aplicar dichas operaciones dentro de una ventana deslizante aplicada sobre el flujo de datos de entrada.

# Arquitectura Spark Streaming



**Figura:** Transformaciones sobre ventanas deslizantes.

# Transformaciones DStream en ventanas

- `window(win_len, slide_interv)`: Retorna un DStreams con lotes de DStream de entrada agrupados según la ventana deslizante que se defina.
- `countByWindow(win_len, slide_interv)`: Devuelve el número de elementos que existen en cada ventana aplicada sobre el DStream de entrada.
- `reduceByWindow(func, win_len, slide_interv)`: Retorna un DStream de un único elemento que agrega los elementos dentro de la ventana deslizante aplicada sobre el DStream de entrada, según la operación definida por `func`.

# Transformaciones DStream en ventanas

- `reduceByKeyAndWindow(func, win_len, slide_interv, [num_tasks])`: Como el anterior, pero la agregación se aplica a elementos clave-valor (K,V), aplicación `func` a todos los elementos que comparten la misma clave.
- `reduceByKeyAndWindow(func, inv_func, win_len, slide_interv, [num_tasks])`: Más eficiente que el anterior, calcula incrementalmente el nuevo valor de cada operación `reduce`, agregando los nuevos datos que entran en la ventana según lo indicado por `func` y ejecutando el *inverse reduce* de los datos que abandonan la ventana en cada iteración.
- `countByValueAndWindow(win_len, slide_interv, [num_tasks])`: Aplicada sobre un DStream de elementos (K, V), cuenta en cada ventana el número de elementos que tienen la misma clave, devolviendo pares de tipo (K, Long).



# Programación con DStreams

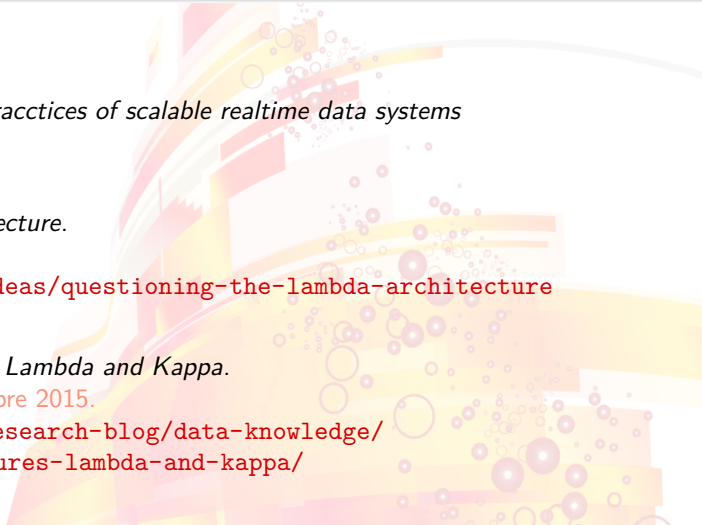



**Ejercicio 3:** ejemplo de función de agregación de valores siguiendo una ventana deslizante y según la clave de cada elemento `reduceByKeyAndWindow(...)`.

- Archivo código Spark Streaming: `networdcount_win.py`.
- Las instrucciones de ejecución son las mismas que en los casos anteriores, pero la salida es diferente.

# Enlaces de interés

- [Spark Streaming programming guide.](#)
- [Diving into Spark Streaming's Execution Model.](#)
- [Code examples Spark Streaming.](#)

# References I

- 
-  [Marz, 2015] Marz, N.  
*Big Data: Principles and best practices of scalable realtime data systems*  
Manning Pub., mayo, 2015.
  
  -  [Kreps, 2014] Kreps, J.  
*Questioning the Lambda Architecture.*  
O'Reilly Radar, julio, 2015.  
<https://www.oreilly.com/ideas/questioning-the-lambda-architecture>
  
  -  [Forgeat, 2015] Forgeat, J.  
*Data Processing Architectures - Lambda and Kappa.*  
Ericsson Research Blog, noviembre 2015.  
<http://www.ericsson.com/research-blog/data-knowledge/data-processing-architectures-lambda-and-kappa/>

# References II



[Feller, 2015] Feller, E.

*Data Processing Architectures - Lambda and Kappa Examples.*

Ericsson Research Blog, noviembre 2015.

[http://www.ericsson.com/research-blog/data-knowledge/  
data-processing-architectures-lambda-and-kappa-examples/](http://www.ericsson.com/research-blog/data-knowledge/data-processing-architectures-lambda-and-kappa-examples/)