UNIVERSIDAD COMPLUTENSE DE MADRID

ntic master School

Spark Structured Streaming

Dr. Pablo J. Villacorta

Septiembre de 2020

Agenda

- Motivación: casos de uso de streaming
- Conceptos básicos y retos del procesamiento en streaming
- Ejemplo de Spark Structured Streaming

1 Motivación y casos de uso

Motivación del procesamiento en streaming

- Es el acto de **incorporar de manera continua nuevos datos** para ir actualizando el resultado del cálculo
  - Los datos de entrada no tienen principio ni fin
  - Los datos son series de eventos que llegan a la aplicación

- El programa calcula **nuevas versiones del resultado** según van llegando nuevos datos
  - Spark automáticamente **actualiza** el resultado

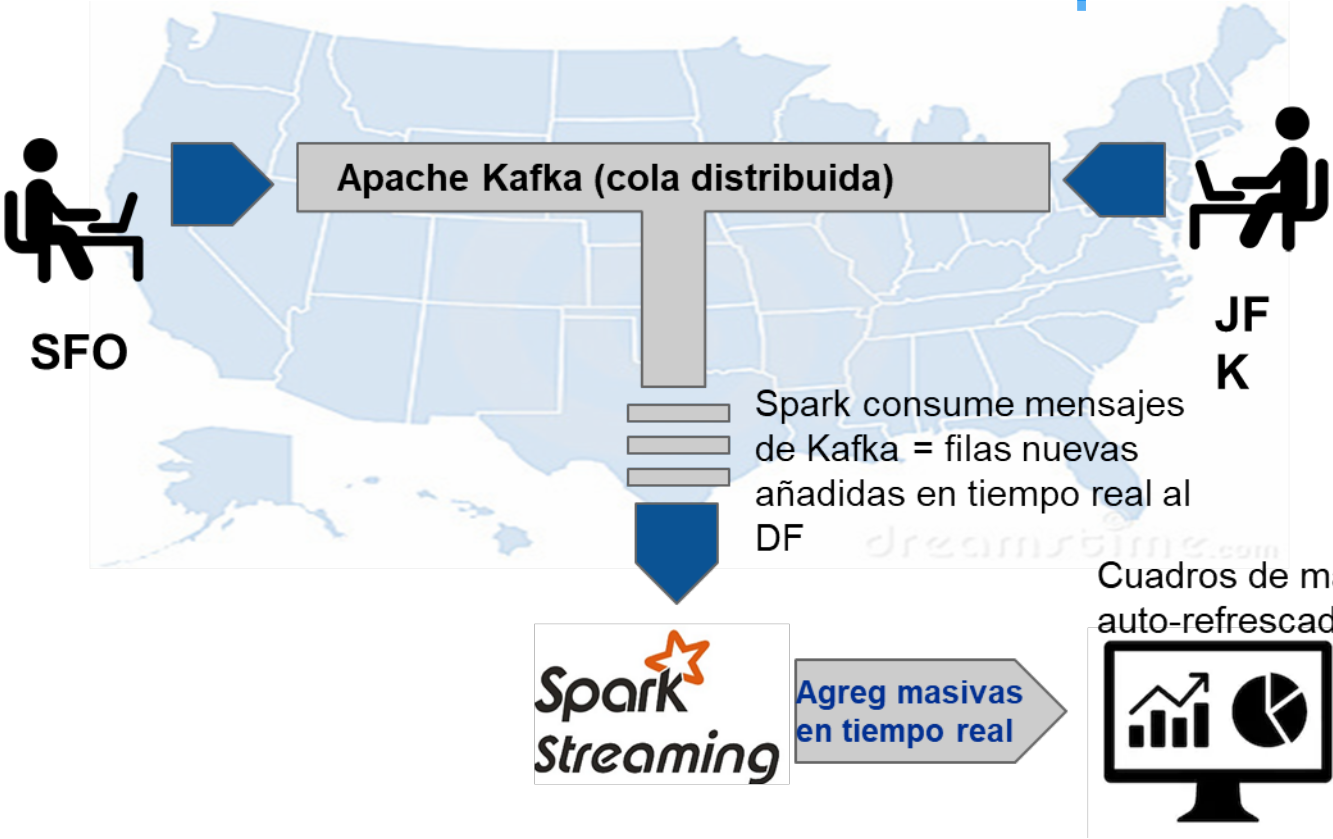
Recordamos los módulos de Spark

- Spark Core: RDDs, operaciones de bajo nivel
- Spark SQL: DataFrames y sintaxis SQL
- Spark MLlib: algoritmos de Machine Learning

- **Spark Streaming**: procesamiento en tiempo real de flujos de datos
  - Dos APIs:
    - DStream API: RDDs (bajo nivel, muy poca ayuda)
    - Structured Streaming: la API Estructurada que ya hemos usado (DataFrames)

- GraphX: procesamiento de grafos

Info de vuelos analizada en tiempo real



Casos de uso para streaming

- Alertas: un nuevo dato causa que se actualice cierto resultado y a consecuencia, se envíe una alerta a una persona
- Reporting en tiempo real: cuadros de mando construidos sobre agregaciones de datos que se actualizan en tiempo real
- ETL incremental: “*mi trabajo batch pero en streaming*”. Limpiar y procesar datos en crudo según van llegando, para escribirlos a un data warehouse.
- Decisiones en tiempo real: ¿es fraudulenta esta transacción? Denegar sobre la marcha en caso de serlo
  - Usando reglas hardcodeadas, o un modelo de ML

Retos de los sistemas de streaming

- Baja latencia (respuesta rápida a un solo evento)
- Alto throughput (procesar rápidamente un volumen alto de entrada. Se mide en salidas por unidad de t). Opuesto a baja latencia habitualmente.
- Procesar datos desordenados (recibidos en orden diferente al que se generaron)
- Mantener un estado interno para determinar qué está pasando
- Ser capaz de unir datos en streaming con datos batch históricos
- Robustez pero sin duplicar salidas

Fuentes de datos habituales en streaming

- Kafka: cola distribuida a la que se le añaden mensajes en un lado y se consumen en el otro
- HDFS: si un proceso externo va añadiendo archivos nuevos a un directorio, Spark puede ir leyendo y procesando continuamente de ahí

API de Spark Streaming (DStreams) - obsoleta

- DStreams API (~obsoleta) : RDDs y ***microbatches***
  - Los datos que van llegando se van **acumulando en una pequeña ventana** (e.g. 500 ms) para formar un pequeño RDD que se procesa en paralelo con Spark, **como trabajo batch**

- No es un verdadero sistema de procesamiento en tiempo real
  - Más latencia, pero a cambio también **mejor throughput**!
  - Los sistemas de procesamiento continuo tienen baja latencia pero mayor sobrecarga por registro: peor cuando recibimos muchos datos de entrada casi a la vez (**bajo throughput**)

API de Structured Streaming

- Siguen siendo **microbatches** (aunque hay una propuesta para soportar verdadero procesamiento continuo, aún sin hacer)

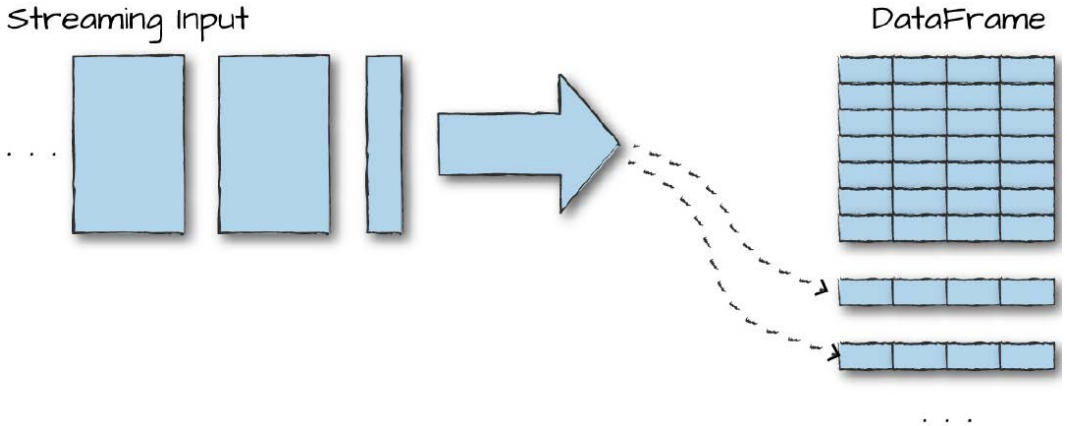
- Idea: misma API que la API Estructurada (DataFrames)
  - *El mismo código batch, transformado a streaming!*

- Conceptualmente, un DataFrame al que **se le añaden filas** en tiempo real
  - El mismo código que funciona para un trabajo batch debería servir sin cambios en Structured Streaming

- Restricción: solo hay **una acción** disponible: **arrancar un flujo** que permanecerá ejecutando continuamente dando resultados

API de Structured Streaming

- Idea: misma API que la API Estructurada (DataFrames)
  - *El mismo código batch, transformado a streaming!*
- Conceptualmente, un DataFrame al que **se le añaden filas** en tiempo real



Ejemplo de Structured Streaming

- Imaginemos que los aeropuertos envían en tiempo real información a nuestro sistema sobre cada vuelo que aterriza
  - Se podrían utilizar distintas arquitecturas para esto. La más habitual sería que Spark leyese en tiempo real desde Kafka, una cola de mensajes distribuida y replicada donde los aeropuertos van escribiendo información
- Para simplificar nuestro notebook, asumimos que un proceso externo crea nuevos archivos en tiempo real en una carpeta de HDFS.
- Spark leerá continuamente esa carpeta y actualizará el resultado de cierta agregación
  - Spark es capaz de incrementalizar automáticamente el cálculo

Ejemplo de Structured Streaming

```
flightsSchema = StructType([
     StructField("Origin", StringType(), True),
     ...]) # en Structured Streaming el esquema es obligatorio

streamingFlights = spark.readStream.schema(flightsSchema)
     .option("maxFilesPerTrigger", 1) # leer un solo fichero en cada operación
     .csv("/data/flightsFolder")

# Operación de agregación como haríamos con cualquier DataFrame
countStreamingDF = streamingFlights.groupBy("Origin", "Dest").count()

# Escribimos periódicamente el resultado. Para probar, escribimos a memoria
countQuery = countStreamingDF
               .writeStream.queryName("countsPerAirport")\
               .format("memory")\
               .outputMode("complete")\ # escribir la salida completa cada vez
               .start()                 # arrancamos el flujo de datos

countQuery.awaitTermination() # obligatorio para evitar que el driver finalice
```

Modos de escritura

Cuando escribimos resultados en tiempo real, hay varias opciones:

- Añadir (solo se añaden nuevos registros al datasource destino)
- Actualizar (modificar solo los registros que han cambiado)
- Completo (re-escribir la salida completa)

Dependiendo del tipo de transformación que hagamos, algunos de estos modos no tienen sentido

- En nuestro ejemplo anterior, *append* no tendría sentido porque los resultados de la agregación deben actualizarse, y sigue habiendo el mismo número de registros

Event-time & watermarks

Funcionalidad de Structured Streaming no disponible en DStreams

- Podemos decir a Spark qué columna de nuestros corresponde al **event-time**: instante en el cual se **generó** el evento
- No importa con cuánto retraso nuestro sistema reciba ese evento
- Permite que Spark procese eventos que llegan desordenados

Watermarks: cómo de *tarde* esperamos ver un evento. Útil cuando queremos, p. ej., agregar eventos generados durante cierta *ventana temporal* definida sobre el event-time.