

UD 6 - Apache Spark - Spark Streaming

1. Stream Processing

Antes de empezar a estudiar Spark Streaming, es recomendable hacer una pequeña introducción a Stream Processing

1.1 ¿Qué es el Stream Processing?

El stream processing está basado en la idea de **procesar los datos de forma continua**. En cuanto estos datos están disponibles se procesan de manera secuencial. Para ello, se usan flujos de datos infinitos y sin límites de tiempo. La manera tradicional de procesar los datos ha sido en batches, agrupados en grandes lotes, esta técnica se llama batch processing.

Actualmente, los servicios en tiempo real que usan estos mecanismos de stream processing cada vez tienen más demanda. A través de estas técnicas es posible acelerar la velocidad a la que se obtiene valor de los datos y generar acciones para interaccionar con los clientes con poca latencia.

Generalmente, las latencias que se consideran al hablar de los sistemas de tiempo real o de stream processing son del orden de 10 milisegundos a 1 segundo. En función del caso de uso o el ámbito de su aplicación, estas latencias pueden reducirse, aunque supondrán importantes desafíos.

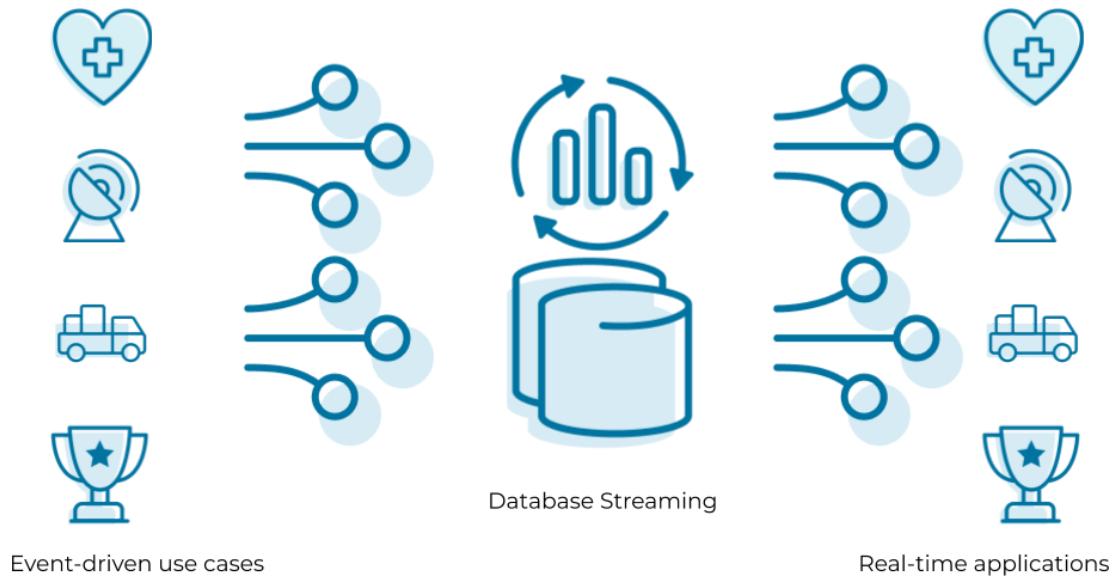


Figura 6.1_Spark Streaming: Streaming Processing. (Fuente: confluent.io)

Los requisitos más importantes a tener en cuenta a la hora de implementar una solución analítica en streaming son los siguientes:

- Cantidad de datos que se deben procesar de forma simultánea (picos de carga)
- Latencias extremo a extremo (end to end)
- Garantías de entrega de mensajes que debe asegurar la solución

1.2 Casos de uso de Streaming

Las fuentes de datos para casos de uso de stream processing pueden estar presentes en cualquier sector, a continuación tienes listados los más populares:

- Monitorización de sistemas, de redes y de aplicaciones
- Dispositivos Internet of Things (IoT)
- Sistemas de recomendación y optimización de resultados
- Transacciones financieras, detección de fraude y trading
- Seguimiento de usuarios en páginas web y comercio electrónico
- Notificaciones en dispositivos y aplicaciones móviles en tiempo real

Los procesos y flujos de trabajo con datos en streaming más usados son **los filtros, la agregación y el enriquecimiento de datos**, es decir, realizar un adecuado **ETL** adaptado a la solución de un entorno real. Esto nos permite reducir la cantidad de información, calcular agregados de datos en ventanas temporales, reducir la cantidad de información persistida, entre otras. También nos permite el enriquecimiento, agregar información a un dataset en tiempo real. Esto ofrece ventajas respecto a enriquecerla cuando ya se encuentra persistida en un almacenamiento, y es que permite tomar decisiones más rápido con los datos.

1.2 Conceptos Básicos en Stream Processing

A continuación vamos a desarrollar algunos conceptos y términos básicos que se han generado alrededor de las tecnologías de stream processing.

Los sistemas de streaming distribuidos tienen tres maneras de gestionar las **garantías de entrega** de los mensajes en sus protocolos:

- **At-least-once:** Garantiza que el mensaje siempre se entregará. Es posible que en caso de fallo se entregue varias veces, pero no se perderá ningún mensaje en el sistema.
- **At-most-once:** Garantiza que el mensaje se entregará una vez o no se entregará. Un mensaje nunca se entregará más de una vez.
- **Exactly-once:** Garantiza que todos los mensajes se van a entregar exactamente una vez, realizando el sistema las comprobaciones necesarias para que esto suceda.

Existen numerosas aplicaciones que no se pueden permitir la existencia mensajes duplicados o perdidos debido a fallos en la comunicación o en las aplicaciones. Por esta razón es tan importante que existan sistemas que garanticen la entrega exactamente una vez como Apache Kafka o Flink.

- **Tupla o evento:** conjunto de elementos o de tipos de datos simples guardados de forma consecutiva. También, los podemos llamar eventos o mensajes. Los eventos representan un cambio de estado en el sistema y normalmente tienen un orden basado en el tiempo.
- **Flujo de datos:** También llamado stream o stream de eventos. Se trata de una secuencia infinita de tuplas o de eventos en la que el orden importa. Este flujo de datos viaja desde los productores hacia los consumidores de datos.
- **Ventanas de procesamiento:** Dividen los datos de entrada en partes finitas. Permiten tratar las secuencias infinitas con unos recursos limitados como la memoria del sistema. Pueden estar basadas en tiempo o en el número de elementos y se pueden desplazar a medida que se procesa su contenido. Existen varios tipos de ventanas dependiendo de las características del sistema.
- **Operaciones con y sin estado:** Las operaciones sin estado permiten obtener un resultado por cada uno de los eventos procesados. Las operaciones con estado operan sobre un conjunto de elementos para generar una salida.

Para mantener la tolerancia a fallos, los sistemas de streaming usan **checkpointing** y no eliminan los eventos del sistema una vez procesados. De esta manera, almacenan de forma persistente el estado del sistema en instantes de tiempo y el punto en el que se encuentran para poder recuperar la información en el caso de que ocurra algún fallo de red o en los propios nodos.

- **Backpressure:** Es el mecanismo que indica a la tecnología que los consumidores no pueden procesar más eventos en un instante concreto. Evita que el sistema se sature cuando se publican eventos a más velocidad de la que se consumen. Normalmente, se implementa un mecanismo de **buffering**. Si se excede su capacidad, se pueden eliminar los eventos con una política definida de tipo LIFO, FIFO, etc. La capacidad de escalar junto a un buen mecanismo de backpressure son esenciales para garantizar la alta disponibilidad y el rendimiento del sistema.

2. Spark Streaming

Spark Streaming es una extensión del core Spark API que permite el procesamiento de flujos de datos en forma continua, escalables, de alto rendimiento y tolerante a fallos. Los datos se pueden ingerir de muchas fuentes, como Kafka, Kinesis o sockets TCP, y se pueden procesar utilizando algoritmos complejos expresados con funciones de alto nivel como *map*, *reduce*, *join and window*. Finalmente, los datos procesados se pueden enviar a sistemas de archivos, bases de datos y paneles de control en tiempo real. De hecho, puede aplicar los algoritmos de procesamiento de gráficos y aprendizaje automático de Spark en flujos de datos.



Figura 6.2_Spark Streaming: Spark Streaming Architecture. (Fuente: spark.apache.org)

Spark tiene **2 soluciones de Streaming**:

- **Spark DStream:** Generación previa del motor Spark Streaming. Está basada en **RDDs**
- **Spark Structured Streaming:** Introducido en Spark 2.0, está basada en el uso de **Dataframe**. Facilita la creación de aplicaciones y canalizaciones en tiempo real usando las

APIs Spark.

2.1 DStream

Internamente funciona de la siguiente manera. Spark Streaming recibe flujos de datos de entrada en tiempo real(stream) y los divide en lotes (batch), que luego son procesados por el motor Spark para generar el flujo final de resultados en lotes.



Figura 6.3_Spark Streaming: Flujo DStream. (Fuente: spark.apache.org)

Spark Streaming proporciona una abstracción de alto nivel llamada **discretized stream** o **DStream**, que representa un flujo continuo de datos. Los DStreams se pueden crear a partir de flujos de datos de entrada de fuentes como Kafka y Kinesis, o aplicando operaciones de alto nivel en otros DStreams. Internamente, un DStream se representa como una secuencia de RDD.

2.2 Spark Structured Streaming

Structured Streaming es un motor de stream processing escalable y tolerante a fallos integrado en el motor Spark SQL. El motor Spark SQL se encargará de ejecutarlo de forma incremental y continua y de actualizar el resultado final a medida que sigan llegando datos de streaming. También se crean canalizaciones y aplicaciones de streaming de baja latencia y coste eficiente.

Puede utilizar la API Dataset/DataFrame en *Scala*, *Java*, *Python* o *R* para expresar agregaciones de transmisión, ventanas de tiempo de evento, join y uniones de stream a batch, etc. Finalmente, el sistema garantiza tolerancia a fallos de extremo a extremo mediante la entrega de mensajes de una sola vez (**exactly-once**) a través de **checkpointing** y logs.

Además, proporciona las mismas API estructuradas (DataFrames y Datasets) que Spark, por lo que no es necesario desarrollar ni mantener (ni aprender) dos stack de tecnología diferentes para batch y streaming. Además, las API unificadas facilitan la migración de sus trabajos Spark por lotes existentes a trabajos de transmisión.

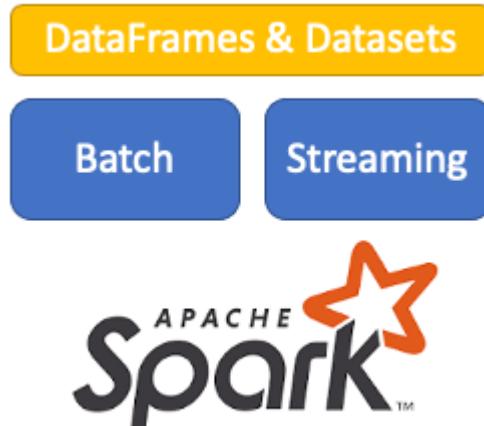


Figura 6.4_Spark Streaming: Streaming Structured batch and stream. (Fuente: spark.apache.org)

Internamente, de forma predeterminada, las consultas de Structured Streaming se procesan utilizando un *micro-batch processing engine*, que procesa flujos de datos como una serie de pequeños trabajos por lotes, logrando así latencias de extremo a extremo tan bajas como 100 milisegundos y garantías de tolerancia a fallos exactly-once.

2.3. Spark Structured Streaming. Programming Model

La idea clave en **Structured Streaming** es tratar un flujo de datos en tiempo real como una tabla que se agrega continuamente. Esto conduce a un nuevo modelo de procesamiento de flujos que es muy similar al modelo de procesamiento batch. Expresará su cálculo de transmisión como una consulta estándar batch como en una tabla estática, y Spark lo ejecutará como una consulta incremental en la tabla de entrada ilimitada.

Si consideramos el flujo de datos de entrada como la "Input Table". Cada elemento de datos que llega a la secuencia es como una nueva fila que se agrega a la tabla de entrada.

Info

En el contexto de Apache Spark, "**unbounded**" se refiere generalmente a datos que no tienen un límite definido en el tiempo, es decir, datos que continúan llegando de manera continua sin un punto final específico. Esto se contrasta con "bounded", que se refiere a conjuntos de datos con un límite claro y definido.

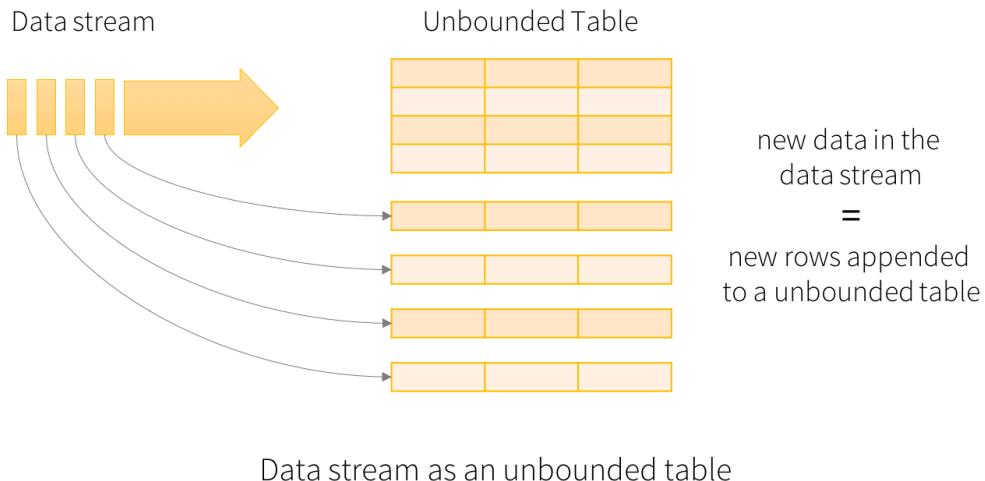
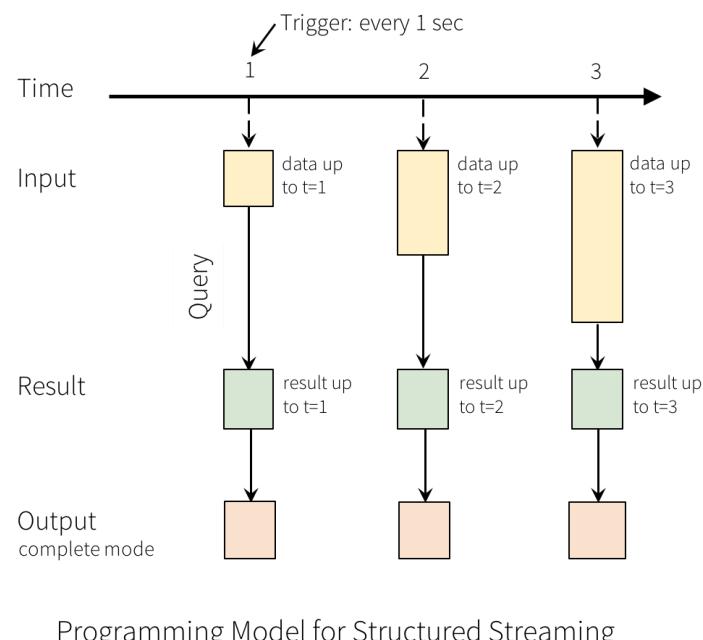


Figura 6.5_Spark Streaming: Streaming Structured - stream as a table. (Fuente: spark.apache.org)

Una consulta sobre la entrada generará la “Result Table”. En cada intervalo de activación (por ejemplo, cada segundo), se agregan nuevas filas a la tabla de entrada, que eventualmente actualiza la tabla de resultados. Cada vez que se actualiza la tabla de resultados, queremos escribir las filas de resultados modificadas en un receptor externo.



Programming Model for Structured Streaming

Figura 6.6_Spark Streaming: Streaming Structured model. (Fuente: spark.apache.org)



Note

En el siguiente punto profundizaremos sobre los "Outputs"

Handling Event-time and Late Data

Event-time (tiempo del evento) es el tiempo incrustado en los datos mismos. Para muchas aplicaciones, es posible que queramos operar en este momento del evento. Por ejemplo, si desea obtener la cantidad de eventos generados por dispositivos IoT cada minuto, entonces probablemente queramos usar la hora en la que se generaron los datos (es decir, la hora del evento en los datos), en lugar de la hora a la que la recibe Spark. Este tiempo de evento se expresa de manera muy natural en este modelo: cada evento de los dispositivos es una fila en la tabla y el tiempo de evento es un valor de columna en la fila.

Esto permite que las agregaciones basadas en ventanas (por ejemplo, número de eventos por minuto) sean solo un tipo especial de agrupación y agregación en la columna de tiempo del evento: cada ventana de tiempo es un grupo y cada fila puede pertenecer a múltiples ventanas/grupos. Por lo tanto, dichas consultas de agregación basadas en ventanas de tiempo de eventos se pueden definir de manera consistente tanto en un conjunto de datos estáticos (por ejemplo, a partir de registros de eventos de dispositivos recopilados) como en un flujo de datos, lo que hace la vida del usuario mucho más fácil.



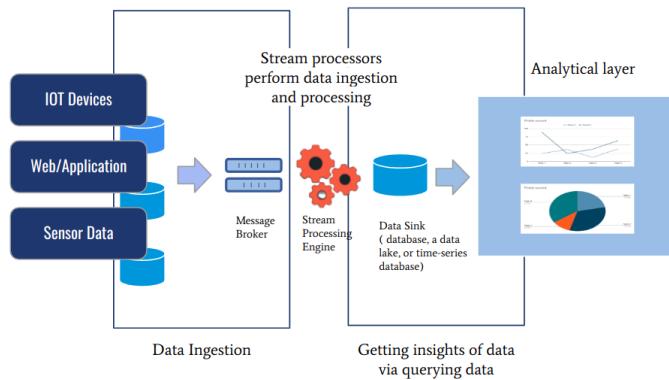
Continuous Processing

Desde Spark 2.3, existe un nuevo modo de ejecución llamado **continuous processing** (procesamiento continuo) que permite una baja latencia de extremo a extremo baja (~1 ms) con garantías de tolerancia a fallas **exactly-once**. Está en fase experimental y la trataremos cuando entre en versión estable

3. Arquitectura Spark Streaming

Una vez vista las 2 soluciones que ofrece Spark Streaming, vamos a intentar dar una pequeña representación de la **Arquitectura y WorkFlow** de Spark Streaming.

Real-Time Processing



© 2022 Copyright Velotio

Figura 6.7_Spark Streaming: Architecture Spark Streaming (Fuente: velotio.com)

Spark Streaming, de forma general, y dejando atrás muchas particularidades:

1. Recepción de datos (**inputs**) en tiempo real de diferentes fuentes (**source**)
2. Estos son procesados por la lógica de procesamiento
3. Los datos de salida (**outputs**) del procesamiento, la envía a diferentes "almacenamientos externos" o sumideros (**sink**) y en un modo (**mode**) determinado. Vamos a dar una pequeña explicación de cada uno de ellos

3.1 Input source

Algunas de las fuentes integradas. Puedes consultarlas todas en [input source](#)

- **File source:** Lee archivos de un directorio como un flujo de datos. Los archivos se procesarán en el orden de modificación del archivo. **Los formatos de archivo admitidos son texto, CSV, JSON, ORC, Parquet.**
- **Socket source (for testing):** Lee datos de texto UTF8 desde una conexión de socket. Esta fuente de datos escuchará el socket especificado e incorporará cualquier dato en Spark Streaming. Se utiliza únicamente para pruebas.
- **Rate source (for testing):** Genera automáticamente datos que incluyen 2 columnas que contienen `timestamp` and `value`. Donde `timestamp` es un tipo de marca de tiempo que contiene la hora de envío del mensaje y `value`, que es Long Type y contiene el recuento de mensajes, comenzando desde 0 como primera fila. Se utiliza únicamente para pruebas.
- **Rate Per Micro-Batch source (for testing):** Genera automáticamente datos especificando el número de filas por micro-batch. Cada fila de salida contiene `timestamp` and `value`.

- **Kafka source:** lee datos de Kafka. Es compatible con las versiones 0.10.0 o superiores de *Kafka broker*. Consulte la [Guía de integración de Kafka](#) para obtener más detalles.

3.2 Output Mode

Output Mode se define como lo que se escribe en el almacenamiento externo. La salida se puede definir en un modo diferente:

- **Complete Mode:** toda la tabla de resultados actualizada se escribirá en el almacenamiento externo. Depende del conector de almacenamiento decidir cómo manejar la escritura en toda la tabla.
- **Append Mode:** solo las nuevas filas agregadas en la tabla de resultados desde el último **trigger** se escribirán en el almacenamiento externo. Esto solo se aplica a las consultas en las que no se espera que cambien las filas existentes en la tabla de resultados.
- **Update Mode:** solo las filas que se actualizaron en la tabla de resultados desde el último **trigger** se escribirán en el almacenamiento externo (disponible desde Spark 2.1.1). Debemos tener en cuenta que esto es diferente del *complete mode* en que este modo solo genera las filas que han cambiado desde el último **trigger**. Si la consulta no contiene agregaciones, será equivalente al *append mode*.

i Info

Hay que tener en cuenta que cada modo es aplicable a ciertos tipos de consultas. Esto se analiza en detalle más adelante.

3.3 Output Sinks

Output Sinks indican a qué tipo de almacenamiento externo se envían los datos procesados.

- **File sink:** Almacena el contenido del Datafram en un fichero dentro de un directorio. Los tipos de archivos admitidos son `csv`, `json`, `orc`, and `parquet`.

```

1  writeStream
2      .format("parquet")          // can be "orc", "json", "csv", etc.
3      .option("path", "path/to/destination/dir")
4      .start()

```

- **Kafka sink:** Publica los datos en uno o mas topics en Kafka.

```

1  writeStream
2      .format("kafka")
3      .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
4      .option("topic", "updates")

```

```
5     .start()
```

- **Foreach sink**: se aplica a cada fila de un DataFrame y se puede usar al escribir lógica personalizada para almacenar datos.

```
1  writeStream
2      .foreach(...)
3      .start()
```

- **Console sink (for debugging)**: Imprime el contenido del DataFrame por consola.

```
1  writeStream
2      .format("console")
3      .start()
```

- **Memory sink (for debugging)**: la salida se almacena en la memoria como una tabla en memoria. Se admiten append y complete mode. Esto debe usarse con fines de depuración en volúmenes de datos bajos, ya que toda la salida se recopila y almacena en la memoria del controlador.

```
1  writeStream
2      .format("memory")
3      .queryName("tableName")
4      .start()
```

Para comprender el modelo, lo veremos paso a paso en el Ejemplo1.

4. Ejemplo 1. Custom Wordcount

Supongamos que queremos mantener un recuento continuo de palabras de los datos de texto recibidos de un servidor de datos que escucha en un **socket TCP**.

Observa con detalle la descripción de la documentación oficial de cada enlace y los métodos que tiene.

1. Tenemos que importar las clases necesarias y crear una `SparkSession` local, el punto de partida de todas las funcionalidades relacionadas con Spark.

```
1  from pyspark.sql import SparkSession
2  from pyspark.sql.functions import explode
3  from pyspark.sql.functions import split
4
5  spark = SparkSession \
6      .builder \
7      .appName("StructuredNetwork_CustomWordCount_BDA") \
8      .getOrCreate()
```

2. Creamos un **streaming DataFrame** que represente datos de texto recibidos de un servidor que escucha en localhost:9999 y transformemos el DataFrame para calcular el recuento de palabras.
3. Source (fuente de datos): Creamos un flujo de lectura con `readStream`. Como puedes ver en la documentación, éste devuelve un `DataStreamReader` que será usado para crear un streaming DataFrame.
4. Indicamos el formato tipo socket y el resto de configuración y lo cargamos con `load()` para que devuelva el DataFrame

```

1 # Create DataFrame representing the stream of input lines from connection
2 to localhost:9999
3 lines = spark \
4     .readStream \
5     .format("socket") \
6     .option("host", "localhost") \
7     .option("port", 9999) \
8     .load()

```

3. Realizamos ahora la lógica de procesamiento en cualesquieras de las APIs Spark(Dataframe API y/o Spark SQL)
4. `lines` va a contener ahora el DataFrame resultante del texto recogido en streaming.
5. Partimos las palabras de cada línea por un espacio
6. Contamos las palabras agrupando por el alias `word`

```

1 # Split the lines into words
2 words = lines.select(
3     explode(
4         split(lines.value, " ")
5     ).alias("word")
6 )
7
8 # Generate running word count
9 wordCounts = words.groupBy("word").count()

```

4. Configuramos la salida de los datos ya procesados (**sink**) con `writeStream` que devuelve un `DataStreamWriter`
 - a. Configuramos un `output sink` por consola
 - b. Configuramos un `output mode` complete
 - c. Iniciamos con `start()`

```

1 # Start running the query that prints the running counts to the console
2 query = wordCounts \
3     .writeStream \
4     .outputMode("complete") \

```

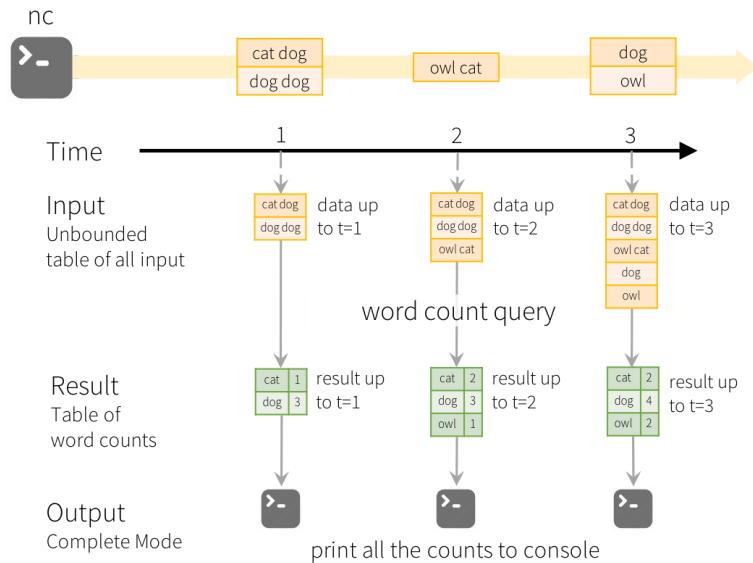
```
5     .format("console") \  
6     .start()
```

5. Por último, Dejamos a la con `awaitTermination()`

```
1 # Waiting  
2 query.awaitTermination()
```

6. La aplicación completa sería:

```
1 from pyspark.sql import SparkSession  
2 from pyspark.sql.functions import explode  
3 from pyspark.sql.functions import split  
4  
5 spark = SparkSession \  
6     .builder \  
7     .appName("StructuredNetwork_CustomWordCount_BDA") \  
8     .getOrCreate()  
9  
10  
11 # Create DataFrame representing the stream of input lines from connection  
to localhost:9999  
12 lines = spark \  
13     .readStream \  
14     .format("socket") \  
15     .option("host", "localhost") \  
16     .option("port", 9999) \  
17     .load()  
18  
19 # Set Spark logging level to ERROR to avoid various other logs on  
console.  
20 spark.sparkContext.setLogLevel("ERROR")  
21  
22 # Split the lines into words  
23 words = lines.select(  
24     explode(  
25         split(lines.value, " ")  
26     ).alias("word")  
27 )  
28  
29 # Generate running word count  
30 wordCounts = words.groupBy("word").count()  
31  
32 # Start running the query that prints the running counts to the console  
33 query = wordCounts \  
34     .writeStream \  
35     .outputMode("complete") \  
36     .format("console") \  
37     .start()  
38  
39 # Waiting  
40 query.awaitTermination()
```



Model of the Quick Example

Figura 6.8_Spark Streaming: Model Ejemplo 1 (Fuente: spark.apache.org)

- Ejecutamos el comando NetCat en el puerto 9999 que es donde nuestra app estará escuchando

```
1 nc -l k 9999
```

- Ejecutamos nuestra app de Spark Streaming

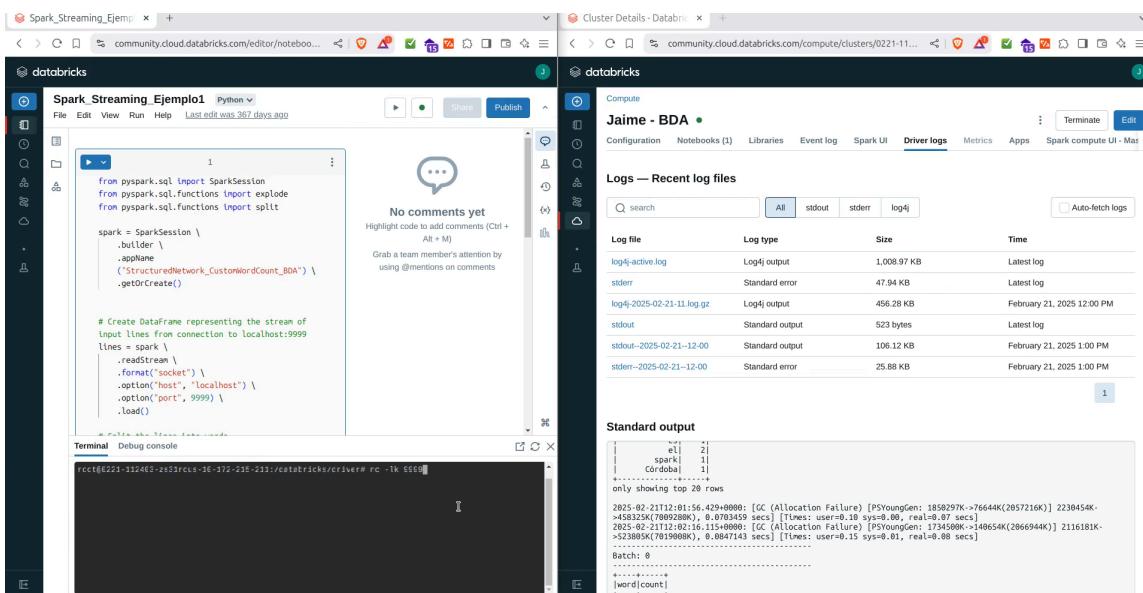
✓ Ejecución

Puedes realizar el ejercicio tanto en Databricks como en nuestro propio cluster. Para DataBricks debes tener activo "Web Terminal" en la configuración de usuario: `Usuario -> Settings -> Workspace admin -> Compute -> Web Terminal (Enabled)`. Para ver la salida, accede a `Driver logs -> Standard output` (No podrás verlo en tiempo real, tendrás que ir actualizando, a diferencia de en nuestro propio cluster Spark).

En nuestro cluster, puedes tanto ejecutarlo con `spark-submit` o `pyspark`, ya sea como shell o levantando jupyter

- Ejecutado en Databrick

Figura 6.9_Spark Streaming: Ejemplo 1 Databrick



Animación 6.1_Spark Streaming: Ejemplo 1 Databrick

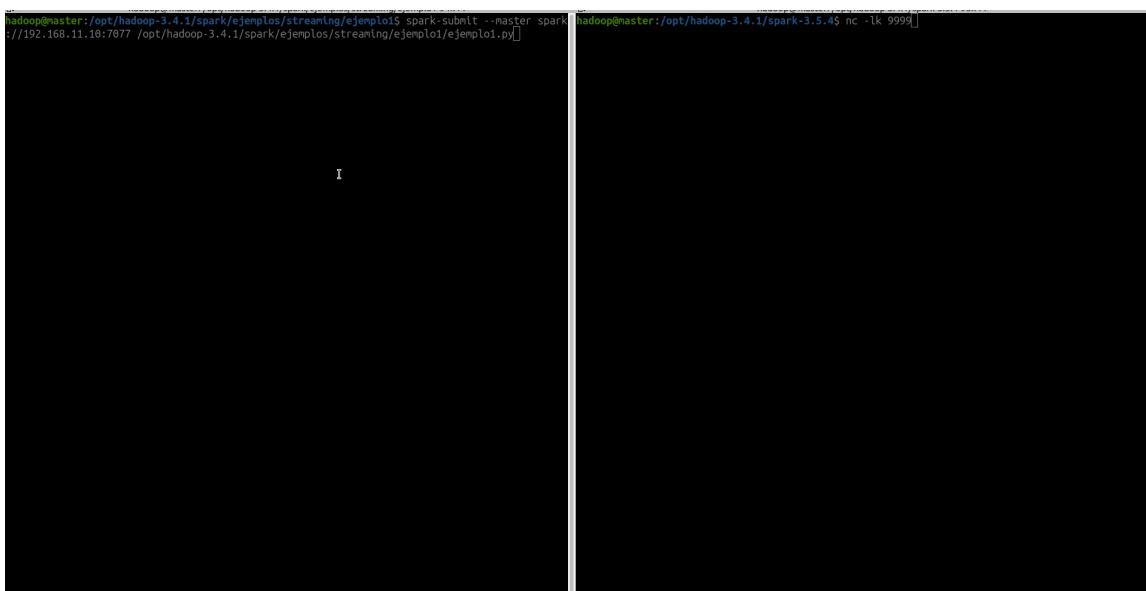
10. Ejecutado en nuestro cluster

```

hadoop@master:/opt/hadoop-3.3.6/spark-3.5.0$ hadoop@master:/opt/hadoop-3.3.6/spark-3.5.0$ nc -lk 9999
24/02/20 10:04:32 INFO TaskSchedulerImpl: Killing all running tasks in stage 15: Stage finished
24/02/20 10:04:32 INFO DAGScheduler: Job 7 finished: start at NativeMethodAccessorImpl.java:10, took 3,38
2967 ms
24/02/20 10:04:32 INFO WriteToDataSourceV2Exec: Data source write support MicroBatchWrite[epoch: 7, writ
er: ConsoleWriter[numRows=20, truncate=true]] is committing.
Batch: 7
-----
word|count|
-----+
en| 1|
Estamos| 1|
En| 2|
ejemplo| 1|
palabras| 1|
BDA| 1|
los| 1|
streaming| 1|
Hola| 1|
clase| 1|
cuenta| 1|
el| 2|
correctamente| 1|
Probando| 1|
de| 4|
es| 1|
el| 2|
Capitán| 1|
spark| 1|
nueve| 1|
-----+
only showing top 20 rows
24/02/20 10:04:32 INFO WriteToDataSourceV2Exec: Data source write support MicroBatchWrite[epoch: 7, writ
er: ConsoleWriter[numRows=20, truncate=true]] committed.

```

Figura 6.10_Spark Streaming: Ejemplo 1 cluster



Animación 6.2_Spark Streaming: Ejemplo 1 cluster

5. Ejemplo2. Usando Rate Source

Vamos a usar `ratesource` y `consolesink`. Rate Source generará automáticamente datos que luego imprimiremos en consola.

1. Tenemos que importar las clases necesarias y crear una `SparkSession` local, el punto de partida de todas las funcionalidades relacionadas con Spark.

```

1  from pyspark.sql import SparkSession
2  from pyspark.sql.functions import *
3
4  spark = SparkSession \
5      .builder \

```

```

6   .appName("Ejemplo2_BDA") \
7   .getOrCreate()

```

2. Establecemos el nivel de registro en `Error` para evitar los logs de `Warning` e `INFO`.

```

1 # Set Spark logging level to ERROR to avoid various other logs on console.
2 spark.sparkContext.setLogLevel("ERROR")

```

3. Creamos un **streaming DataFrame** usando `rate source`. Especificamos el formato `rate` y las filas por segundo a 1 para generar 1 fila para cada microbatch y cargar los datos en el DataFrame de transmisión de `df_init`. Además, verificamos si `df_init` es un DataFrame de transmisión o no.

```

1 df_init = spark \
2     .readStream \
3     .format("rate") \
4     .option("rowsPerSecond", 1) \
5     .load()
6
7 print("Streaming DataFrame : " + str(df_init.isStreaming))

```

La salida debería decir lo siguiente:

```
1 Streaming DataFrame : true
```

4. Lógica de procesamiento: Realizamos una transformación básica `df_init` para generar otra columna `result` simplemente sumando `1` a la columna `value`:

```

1 df_result = df_init \
2     .withColumn("result", col("value") + lit(1))

```

5. Configuramos la salida de los datos ya procesados. Usamos `append mode` para generar solo los datos recién generados y usamos `console sink` para mostrarlos por consola.

```

1 df_result \
2     .writeStream \
3     .outputMode("append") \
4     .option("truncate", False) \
5     .format("console") \
6     .start() \
7     .awaitTermination()

```

6. La aplicación completa sería:

```

1 from pyspark.sql import SparkSession
2 from pyspark.sql.functions import *
3
4 spark = SparkSession \
5     .builder \

```

```

6     .appName("Ejemplo2_BDA") \
7     .getOrCreate()
8
9 # Set Spark logging level to ERROR to avoid various other logs on
10    console.
11
12 df_init = spark \
13     .readStream \
14     .format("rate") \
15     .option("rowsPerSecond", 1) \
16     .load()
17
18 print("Streaming DataFrame : " + str(df_init.isStreaming))
19
20 df_result = df_init \
21     .withColumn("result", col("value") + lit(1))
22
23 df_result \
24     .writeStream \
25     .outputMode("append") \
26     .option("truncate", False) \
27     .format("console") \
28     .start() \
29     .awaitTermination()

```

7. Salida

```

1 -----
2 Batch: 1
3 -----
4 +-----+-----+
5 |timestamp          |value|result|
6 +-----+-----+
7 |2025-02-21 12:21:19.606|0     |1      |
8 |2025-02-21 12:21:20.606|1     |2      |
9 +-----+-----+
10
11 -----
12 Batch: 2
13 -----
14 +-----+-----+
15 |timestamp          |value|result|
16 +-----+-----+
17 |2025-02-21 12:21:21.606|2     |3      |
18 |2025-02-21 12:21:23.606|4     |5      |
19 |2025-02-21 12:21:22.606|3     |4      |
20 +-----+-----+
21
22 -----
23 Batch: 3
24 -----
25 +-----+-----+
26 |timestamp          |value|result|
27 +-----+-----+
28 |2025-02-21 12:21:24.606|5     |6      |

```

```

29 +-----+-----+
30 |-----+-----+
31 |-----+-----+
32 Batch: 4
33 |-----+-----+
34 |-----+-----+
35 |timestamp | value|result|
36 |-----+-----+
37 |2025-02-21 12:21:25.606|6 |7 |
38 |-----+-----+

```

8. Ejecución en nuestro cluster y Databrick

The screenshot shows a terminal window on the left and a Databricks interface on the right.

Terminal Output:

```

Batch: 1
+-----+-----+
|timestamp | value|result|
+-----+-----+
|2025-02-21 12:21:19.606|0 |1 |
|2025-02-21 12:21:20.606|1 |2 |
+-----+-----+
Batch: 2
+-----+-----+
|timestamp | value|result|
+-----+-----+
|2025-02-21 12:21:21.606|2 |3 |
|2025-02-21 12:21:23.606|4 |5 |
|2025-02-21 12:21:22.606|3 |4 |
+-----+-----+
Batch: 3
+-----+-----+
|timestamp | value|result|
+-----+-----+
|2025-02-21 12:21:24.606|5 |6 |
+-----+-----+
Batch: 4
+-----+-----+
|timestamp | value|result|
+-----+-----+
|2025-02-21 12:21:25.606|6 |7 |
+-----+-----+
Batch: 5
+-----+-----+
|timestamp | value|result|

```

Databricks Driver logs:

The Databricks interface shows the "Driver logs" tab selected. It displays a list of recent log files corresponding to the batches shown in the terminal. Each log file entry includes a timestamp, value, and result.

- Batch: 3

timestamp	value	result
2025-02-21 12:22:26.873	2	3
- Batch: 4

timestamp	value	result
2025-02-21 12:22:27.873	3	4
- Batch: 5

timestamp	value	result
2025-02-21 12:22:28.873	4	5
- Batch: 6

timestamp	value	result
2025-02-21 12:22:29.873	5	6
- Batch: 7

timestamp	value	result
2025-02-21 12:22:30.873	6	7
- Batch: 8

timestamp	value	result
2025-02-21 12:22:31.873	7	8

Figura 6.11_Spark Streaming: Ejecución Ejemplo 2

6. Ejemplo 3. Stock Exchange Data

Vamos a usar `file` y `consoleSink`.

✓ File System Databricks

Puedes realizar el ejercicio tanto en Databricks como en nuestro propio cluster. Para DataBricks necesitas acceder al DBFS: Catalog -> DBFS -> FileStore -> shared_uploads -> tu_usuario o la ruta que tu tengas. Desde aquí upload y arrastráis la carpeta. **Spark Streaming no lee archivos que ya estuvieran en el directorio, sólo los nuevos que van llegando**

Para poder usar DBFS hay que activar **DBFS File Browser**. Para ello debes activar "DBFS File Browser" en la configuración de usuario: Usuario -> Admin settings -> Workspace setting -> DBFS File Browser (Enabled)

En nuestro cluster, sólo tienes que indicar la ruta del directorio de los ficheros para su lectura en streaming

1. Preparación del ejemplo. Para este ejemplo, dentro del directorio he creado dos carpetas: data y stream, para ir añadiendo los ficheros de la carpeta data a stream y spark streaming los vaya leyendo.
2. Vamos a usar como base el [siguiente dataset](#) de datos de precios diarios para índices que rastrean las bolsas de valores de todo el mundo de kaggle
3. Lo he partido en 4 partes para poder realizar la lectura de streaming de ficheros de un directorio

```
1 wget https://gist.githubusercontent.com/jaimerabasco/6a82b19a4f7aa3d855e4ae9d9e38df97/
2 wget https://gist.githubusercontent.com/jaimerabasco/6a82b19a4f7aa3d855e4ae9d9e38df97/
3 wget https://gist.githubusercontent.com/jaimerabasco/6a82b19a4f7aa3d855e4ae9d9e38df97/
4 wget https://gist.githubusercontent.com/jaimerabasco/6a82b19a4f7aa3d855e4ae9d9e38df97/
```

4. Código del programa

- a. Creamos el schema
- b. Leemos en el directorio los ficheros entrantes (input files)
- c. Agrupamos por indice bursátil y valor más alto por año
- d. Mostramos en consola
- e. Prueba Output mode update y complete
- f. Está realizado en Dataframe API
- g. Comentada hay una versión Spark SQL

6.1 Databricks

```

Spark_Streaming_Ejemplo3_BDA

1  from pyspark.sql import SparkSession
2  from pyspark.sql.functions import *
3  from pyspark.sql.types import *
4
5  spark = SparkSession \
6      .builder \
7      .appName("Spark_Streaming_Ejemplo3_BDA") \
8      .getOrCreate()
9
10 # Set Spark logging level to ERROR to avoid various other logs on console
11 spark.sparkContext.setLogLevel("ERROR")
12
13 schema = StructType([
14     StructField("Index", StringType(), True),
15     StructField("Date", DateType(), True),
16     StructField("Open", DoubleType(), True),
17     StructField("High", DoubleType(), True),
18     StructField("Low", DoubleType(), True),
19     StructField("Close", DoubleType(), True),
20     StructField("Adj Close", DoubleType(), True),
21     StructField("Volume", LongType(), True)
22 ])
23
24 # option("maxFilesPerTrigger",2) => This will read maximum of 2 files per
However, it can read less than 2 files.
25 df_init = spark \
26     .readStream \
27     .format("csv") \
28     .option("maxFilesPerTrigger",2) \
29     .option("header", True) \
30     .option("path",
"dbfs:/FileStore/shared_uploads/jaimerabasco@iesgrancapitan.org/spark_streaming"
\)
31     .schema(schema) \
32     .load()
33
34 df_result = df_init.groupBy(col("Index"), year(col("Date")).alias("Year"))
35     .agg(max("High").alias("Max"))
36
37 df_init.printSchema()
38
39 # Registraremos el DataFrame como una vista temporal
40 #df_init.createOrReplaceTempView("stockView")
41
42
43 # Ejecutar la consulta SQL
44 #query = """
45 #SELECT year(Date) AS Year, Index, max(High) AS Max
46 #FROM stockView
47 #GROUP BY Index, Year
48 #"""
49 #df_result = spark.sql(query)

```

```

50
51
52 #df_result = df_init \
53 #     .withColumn("result", col("value") + lit(1))
54
55 # Intenta update y complete mode para entender la diferencia
56 df_result \
57     .writeStream \
58     .outputMode("update") \
59     .option("truncate", False) \
60     .option("numRows", 3) \
61     .format("console") \
62     .start() \
63     .awaitTermination()
64
65 # .option("numRows", 3) => Indica el número de filas que muestra por cons

```

5. Una vez ejecutado el programa se queda esperando la entrada de ficheros en el directorio indicado.
6. Realizar el ejercicio en Databricks lleva asociado algunas acciones añadidas:
 - a. Al no tener acceso a **Databricks CLI** desde web terminal no podemos mover los archivos desde la terminal con `databricks fs cp dbfs:/<ruta_local> dbfs:/<ruta_destino>`
 - b. Por tanto, tendremos que hacerlo desde el notebook. Para ello usaremos `dbutils.fs`
 - c. Como el notebook del ejercicio, una vez que lo lancemos, va a estar escuchando, no podemos ejecutar otro código en el mismo notebook. Por tanto, crearemos otro donde ejecutaremos el código para preparar el entorno e ir copiando los ficheros del directorio `data` a `stream`
 - d. Para ello iremos ejecutando el siguiente código a medida que lo vayamos necesitando. Es importante el orden, ya que **no debemos lanzar la aplicación sin limpiar antes de ficheros el directorio stream**

Spark_Streaming_Ejemplo3_BDA_stream

```

1 #Creamos el directorio stream, si no lo estuviera. SOLO LA PRIMERA VEZ
2 dbutils.fs.mkdirs("dbfs:/FileStore/shared_uploads/jaimerabasco@iesgrancapitan")
3
4 # Cada vez que queramos probarlo, realizamos:
5
6 # Limpiamos el directorio stream ANTES DE LANZAR EL EJEMPLO
7 dbutils.fs.rm("dbfs:/FileStore/shared_uploads/jaimerabasco@iesgrancapitan")
8 dbutils.fs.rm("dbfs:/FileStore/shared_uploads/jaimerabasco@iesgrancapitan")
9 dbutils.fs.rm("dbfs:/FileStore/shared_uploads/jaimerabasco@iesgrancapitan")
10 dbutils.fs.rm("dbfs:/FileStore/shared_uploads/jaimerabasco@iesgrancapitan")
11
12 #UNA VEZ LANZADO EL EJEMPLO, vamos copiando uno a uno mientras observamos
13 # Primera parte del dataset

```

```

14 dbutils.fs.cp("dbfs:/FileStore/shared_uploads/jaimerabasco@iesgrancapitan
15 csv")
16 # Segunda parte del dataset
17 dbutils.fs.cp("dbfs:/FileStore/shared_uploads/jaimerabasco@iesgrancapitan
18 csv")
19 # Tercera parte del dataset
20 dbutils.fs.cp("dbfs:/FileStore/shared_uploads/jaimerabasco@iesgrancapitan
21 csv")
22 # Cuarta parte del dataset
23 dbutils.fs.cp("dbfs:/FileStore/shared_uploads/jaimerabasco@iesgrancapitan
24 csv")

```

7. Seguidos los pasos obtenemos la siguiente salida

The screenshot shows the Databricks UI for a job titled "Jaime - BDA". The "Driver logs" tab is selected. The logs are displayed in sections for different batches (Batch: 0, Batch: 1, Batch: 2, Batch: 3). Each batch section shows a table with columns "Index" and "Year". The logs also include GC allocation failure messages and timing information.

```

Logs — Recent log files

Batch: 0
-----
|Index|Year|
+----+---+
|N/A |192211668|39963| |
|N/A |193211668|42999|
|N/A |1971|610|73999|
+----+---+
only showing top 3 rows

2025-02-21T12:40:40.796+0000: [GC (Allocation Failure) [PSYoungGen: 1824@30K->210193K(2026496K)] 2007321K->393492K(6978560K), 0.1787691 secs] [Times: user=0.33 sys=0.00, real=0.18 secs]
2025-02-21T12:40:45.234+0000: [GC (Allocation Failure) [PSYoungGen: 1794321K->338320K(1992192K)] 1977628K->521627K(6944256K), 0.1211872 secs] [Times: user=0.20 sys=0.00, real=0.12 secs]

Batch: 1
-----
|Index|Year|Max|
+----+---+---+
|HSI |2089|133484|87813|
|GSPTE |2089|11816|29981|
|B00901_SS|2089|3478|01001|
+----+---+---+
only showing top 3 rows

2025-02-21T12:40:57.834+0000: [GC (Allocation Failure) [PSYoungGen: 1907880K->293880K(1862144K)] 2098395K->476392K(6814288K), 0.1211835 secs] [Times: user=0.20 sys=0.01, real=0.13 secs]
2025-02-21T12:41:02.469+0000: [GC (Allocation Failure) [PSYoungGen: 1861850K->416334K(1917952K)] 2045157K->3596514K(6870816K), 0.1789280 secs] [Times: user=0.30 sys=0.00, real=0.10 secs]

Batch: 2
-----
|Index|Year|Max|
+----+---+---+
|SSMI |1995|13327|399902|
|KSII |2012|2057|288029|
|SSMI |2013|2057|980391|
+----+---+---+
only showing top 3 rows

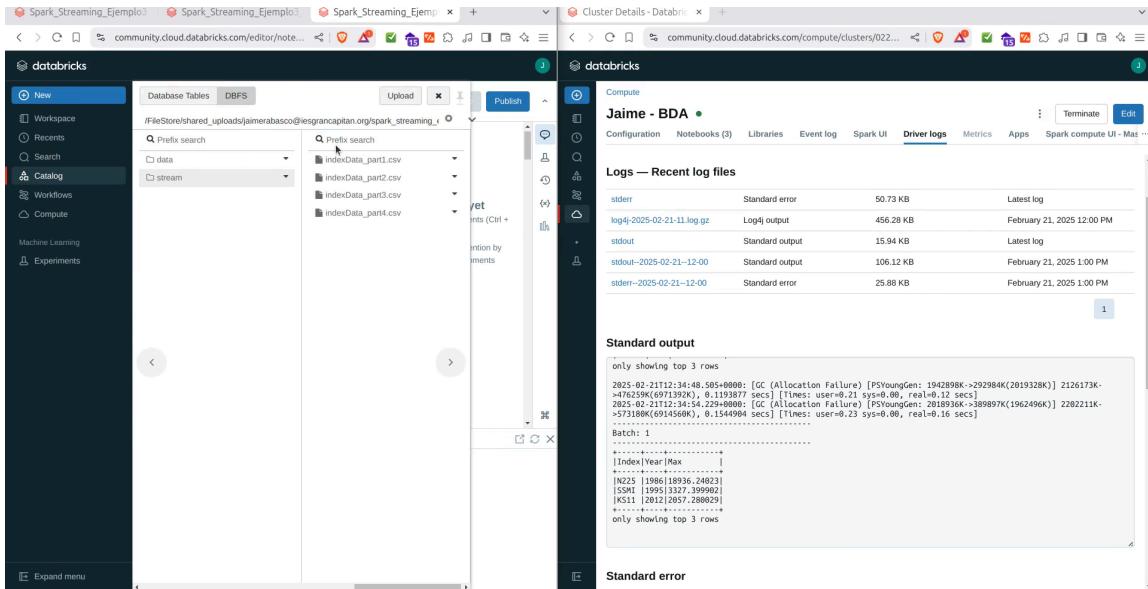
2025-02-21T12:41:24.204+0000: [GC (Allocation Failure) [PSYoungGen: 1874512K->395335K(1853952K)] 2057827K->578650K(6806016K), 0.1438869 secs] [Times: user=0.23 sys=0.00, real=0.14 secs]

Batch: 3
-----
|Index|Year|Max|
+----+---+---+
|N22S |1986|18936|240231|
|N22S |1994|21573|210941|
|N22S |2007|18380|39631|
+----+---+---+
only showing top 3 rows

2025-02-21T12:42:02.916+0000: [GC (Allocation Failure) [PSYoungGen: 1853511K->218704K(1958408K)] 2036826K->402027K(6910464K), 0.1046555 secs] [Times: user=0.19 sys=0.00, real=0.10 secs]

```

Figura 6.12_Spark Streaming: Ejecución Ejemplo 3 Databricks



Animación 6.1_Spark Streaming: Ejemplo 1 Databrick

6.2 Cluster propio

1. Debemos tener en cuenta que tenemos que indicar correctamente el directorio de lectura de ficheros en el código fuente. Para ello, y siguiendo la documentación, hay que indicar si es del sistemas de ficheros, hdfs, S3 `file://`, `hdfs://`, `s3://` ,...
2. Esto lleva asociado una serie de consideraciones importantes. **Todos los workers deben tener acceso al directorio de lectura de ficheros.** Si optamos por la opción del sistema de ficheros, sólo podremos ejecutarlo en el nodo donde se encuentre el directorio, ya que los workers que estén en otro nodo no tendrán acceso al directorio local donde lancemos la app de Spark.
3. Si por el contrario, usamos una fuente distinta (`hdfs://`, `s3://`,...), no tendremos ese problema siempre y cuando todos los workers tengan acceso al directorio, como sería en este caso
4. Vamos a ver, a modo de ejemplo, los 2 casos.

Sistema de ficheros local

1. Código del programa. Para nuestro cluster, vamos a suponer que lo realizamos desde el nodo `master`

`ejemplo3_input_file.py`

```

1  from pyspark.sql import SparkSession
2  from pyspark.sql.functions import *
3  from pyspark.sql.types import *
4
5  spark = SparkSession \

```

```

6     .builder \
7     .appName("Spark_Streaming_Ejemplo3_BDA") \
8     .getOrCreate()
9
10    # Set Spark logging level to ERROR to avoid various other logs on
11    # console.
12    spark.sparkContext.setLogLevel("ERROR")
13
14    schema = StructType([
15        StructField("Index", StringType(), True),
16        StructField("Date", DateType(), True),
17        StructField("Open", DoubleType(), True),
18        StructField("High", DoubleType(), True),
19        StructField("Low", DoubleType(), True),
20        StructField("Close", DoubleType(), True),
21        StructField("Adj Close", DoubleType(), True),
22        StructField("Volume", LongType(), True)
23    ])
24
25    # option("maxFilesPerTrigger",2) => This will read maximum of 2 files per
26    # mini batch. However, it can read less than 2 files.
27    df_init = spark \
28        .readStream \
29        .format("csv") \
30        .option("maxFilesPerTrigger",2) \
31        .option("header", True) \
32        .option("path", "file:///opt/hadoop-
3.4.1/spark/ejemplos/streaming/ejemplo3/stream/") \
33        .schema(schema) \
34        .load()
35
36    df_result = df_init.groupBy(col("Index"),
37        year(col("Date")).alias("Year")) \
38        .agg(max("High").alias("Max"))
39
40    df_init.printSchema()
41
42
43    # Ejecutar la consulta SQL
44    #query = """
45    #SELECT year(Date) AS Year, Index, max(High) AS Max
46    #FROM stockView
47    #GROUP BY Index, Year
48    #"""
49    #df_result = spark.sql(query)
50
51
52    #df_result = df_init \
53    #    .withColumn("result", col("value") + lit(1))
54
55    # Intenta update y complete mode para entender la diferencia
56    df_result \
57        .writeStream \
58        .outputMode("update") \

```

```

59      .option("truncate", False) \
60      .option("numRows", 3) \
61      .format("console") \
62      .start() \
63      .awaitTermination()

```

2. Lanzamos spark master y un worker en el nodo master

```

1  hadoop@master:/opt/hadoop-3.4.1/spark-3.5.4$ ./sbin/start-master.sh
2  hadoop@master:/opt/hadoop-3.4.1/spark-3.5.4$ ./sbin/start-worker.sh
spark://192.168.11.10:7077

```

3. Lanzamos el programa. Recuerda antes borrar todos los ficheros del directorio stream

```

1  spark-submit --master spark://192.168.11.10:7077 /opt/hadoop-
3.4.1/spark/ejemplos/streaming/ejemplo3/ejemplo3_input_file.py

```

4. Copia los **input files** en el directorio stream uno a uno

```

1  hadoop@master:/opt/hadoop-3.4.1/spark/ejemplos/streaming/ejemplo3$ cp
data/indexData_part1.csv stream/
2  hadoop@master:/opt/hadoop-3.4.1/spark/ejemplos/streaming/ejemplo3$ cp
data/indexData_part2.csv stream/
3  hadoop@master:/opt/hadoop-3.4.1/spark/ejemplos/streaming/ejemplo3$ cp
data/indexData_part3.csv stream/
4  hadoop@master:/opt/hadoop-3.4.1/spark/ejemplos/streaming/ejemplo3$ cp
data/indexData_part4.csv stream/

```

5. Salida en consola

```

hadoop@master:/opt/hadoop-3.4.1/spark/ejemplos/streaming/ejemplo3$ ./ejemplo3
+---+
| NYSE | 2021 | 16695.39863 |
| IXIC | 1994 | 864.42993 |
| NVA | 1971 | 618.73999 |
+---+
only showing top 3 rows

Batch: 1
+---+
| Index | Year | Max |
+---+
| HSI | [2018]33484.07813 |
| GSPTSE | [2009]11816.29961 |
| S&P500 | [2009]3478.01081 |
+---+
only showing top 3 rows

Batch: 2
+---+
| Index | Year | Max |
+---+
| S&P500 | [1995]3327.39902 |
| KS11 | [2012]2057.280029 |
| S&P500 | [2015]9537.980391 |
+---+
only showing top 3 rows

Batch: 3
+---+
| Index | Year | Max |
+---+
| N225 | [1996]19936.24923 |
| N225 | [1994]21573.21894 |
| N225 | [2007]18309.39863 |
+---+
only showing top 3 rows

```

```

hadoop@master:/opt/hadoop-3.4.1/spark/ejemplos/streaming/ejemplo3$ cp data/indexData_part1.csv stream/
hadoop@master:/opt/hadoop-3.4.1/spark/ejemplos/streaming/ejemplo3$ cp data/indexData_part2.csv stream/
hadoop@master:/opt/hadoop-3.4.1/spark/ejemplos/streaming/ejemplo3$ cp data/indexData_part3.csv stream/
hadoop@master:/opt/hadoop-3.4.1/spark/ejemplos/streaming/ejemplo3$ cp data/indexData_part4.csv stream/

```

Figura 6.13_Spark Streaming: Ejecución Ejemplo 3 Cluster File

6. Spark UI

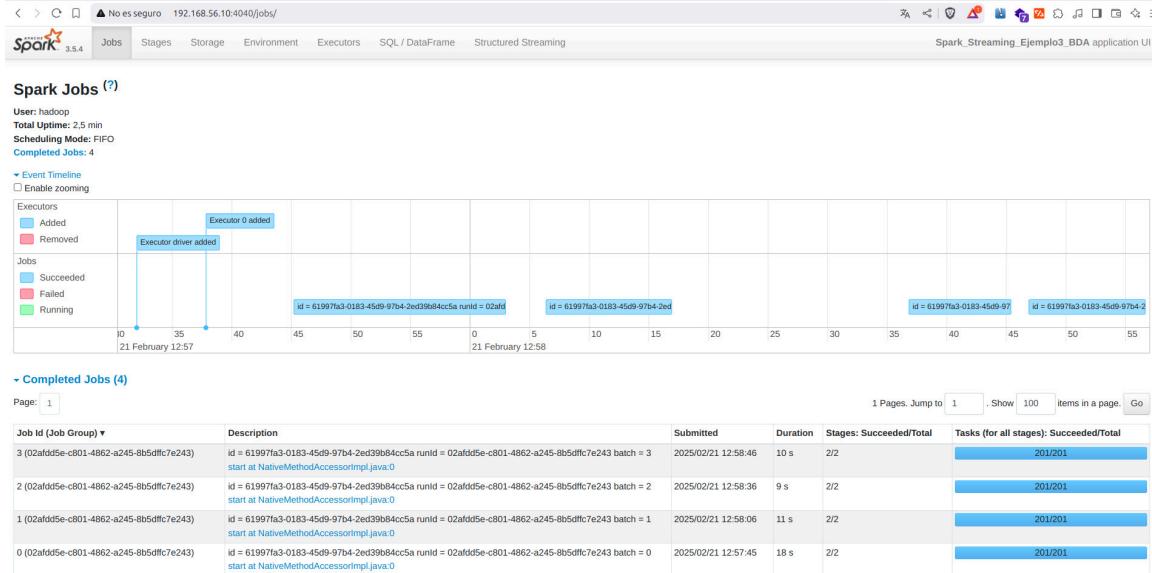


Figura 6.14_Spark Streaming: Ejecución Ejemplo 3 Cluster File Spark UI

HDFS

1. Código del programa. Para nuestro cluster, vamos a suponer que lo realizamos desde el nodo master

```
ejemplo3_input_hdfs.py

1  from pyspark.sql import SparkSession
2  from pyspark.sql.functions import *
3  from pyspark.sql.types import *
4
5  spark = SparkSession \
6      .builder \
7      .appName("Spark_Streaming_Ejemplo3_BDA") \
8      .getOrCreate()
9
10 # Set Spark logging level to ERROR to avoid various other logs on
11 # console.
12 spark.sparkContext.setLogLevel("ERROR")
13
14 schema = StructType([
15     StructField("Index", StringType(), True),
16     StructField("Date", DateType(), True),
17     StructField("Open", DoubleType(), True),
18     StructField("High", DoubleType(), True),
19     StructField("Low", DoubleType(), True),
20     StructField("Close", DoubleType(), True),
21     StructField("Adj Close", DoubleType(), True),
22     StructField("Volume", LongType(), True)
23 ])
24
25 # option("maxFilesPerTrigger",2) => This will read maximum of 2 files per
# mini batch. However, it can read less than 2 files.
26 df_init = spark \
```

```

26     .readStream \
27     .format("csv") \
28     .option("maxFilesPerTrigger",2) \
29     .option("header", True) \
30     .option("path",
31      "hdfs:///bda/spark/ejemplos/streaming/ejemplo3/stream") \
32     .schema(schema) \
33     .load()
34
35 df_result = df_init.groupBy(col("Index"),
36 year(col("Date")).alias("Year")) \
37     .agg(max("High").alias("Max"))
38
39 df_init.printSchema()
40
41 # Registrarmos el DataFrame como una vista temporal
42 #df_init.createOrReplaceTempView("stockView")
43
44 # Ejecutar la consulta SQL
45 #query = """
46 #SELECT year(Date) AS Year, Index, max(High) AS Max
47 #FROM stockView
48 #GROUP BY Index, Year
49 #"""
50
51 df_result = spark.sql(query)
52
53 #df_result = df_init \
54 #    .withColumn("result", col("value") + lit(1))
55
56 # Intenta update y complete mode para entender la diferencia
57 df_result \
58     .writeStream \
59     .outputMode("update") \
60     .option("truncate", False) \
61     .option("numRows", 3) \
62     .format("console") \
63     .start() \
64     .awaitTermination()

```

2. Lanzamos spark master y los workers del **cluster**. Recuerda tener levantado HDFS y los directorios y ficheros ya creados

```

1 hadoop@master:/opt/hadoop-3.4.1/spark-3.5.4$ ./sbin/start-master.sh
2 hadoop@master:/opt/hadoop-3.4.1/spark-3.5.4$ ./sbin/start-workers.sh

```

3. Lanzamos el programa. Recuerda antes borrar todos los ficheros del directorio `stream`

```

1 spark-submit --master spark://192.168.11.10:7077 /opt/hadoop-
3.4.1/spark/ejemplos/streaming/ejemplo3/ejemplo3_input_hdfs.py

```

4. Copia los **input files** en el directorio `stream` uno a uno

```

1 | hdfs dfs -cp
/bda/spark/ejemplos/streaming/ejemplo3/data/indexData_part1.csv
/bda/spark/ejemplos/streaming/ejemplo3/stream
2 | hdfs dfs -cp
/bda/spark/ejemplos/streaming/ejemplo3/data/indexData_part2.csv
/bda/spark/ejemplos/streaming/ejemplo3/stream
3 | hdfs dfs -cp
/bda/spark/ejemplos/streaming/ejemplo3/data/indexData_part3.csv
/bda/spark/ejemplos/streaming/ejemplo3/stream
4 | hdfs dfs -cp
/bda/spark/ejemplos/streaming/ejemplo3/data/indexData_part4.csv
/bda/spark/ejemplos/streaming/ejemplo3/stream

```

5. Salida en consola

```

hadoop@master:/opt/hadoop-3.4.1/spark/ejemplos/streaming/ejemplo3$ hdfs dfs -cp /bda/spark/ejemplos/streaming/ejemplo3/data/indexData_part1.csv /bda/spark/ejemplos/streaming/ejemplo3/stream
hadoop@master:/opt/hadoop-3.4.1/spark/ejemplos/streaming/ejemplo3$ hdfs dfs -cp /bda/spark/ejemplos/streaming/ejemplo3/data/indexData_part2.csv /bda/spark/ejemplos/streaming/ejemplo3/stream
hadoop@master:/opt/hadoop-3.4.1/spark/ejemplos/streaming/ejemplo3$ hdfs dfs -cp /bda/spark/ejemplos/streaming/ejemplo3/data/indexData_part3.csv /bda/spark/ejemplos/streaming/ejemplo3/stream
hadoop@master:/opt/hadoop-3.4.1/spark/ejemplos/streaming/ejemplo3$ hdfs dfs -cp /bda/spark/ejemplos/streaming/ejemplo3/data/indexData_part4.csv /bda/spark/ejemplos/streaming/ejemplo3/stream
hadoop@master:/opt/hadoop-3.4.1/spark/ejemplos/streaming/ejemplo3$ 

Batch: 1
+---+---+
|Index|Year|Max|
+---+---+
|HSII |2018|33484.07813|
|GSPTSE |2009|11816.29981|
|0808001_SS|2009|3478.01001|
+---+---+
only showing top 3 rows

Batch: 2
+---+---+
|Index|Year|Max|
+---+---+
|SMMI |1995|3327.399902|
|KS11 |2012|2057.280029|
|SMMI |2015|9537.903911|
+---+---+
only showing top 3 rows

Batch: 3
+---+---+
|Index|Year|Max|
+---+---+
|N225 |1986|18936.24023|
|N225 |1994|21573.21094|
|N225 |2007|18309.39663|
+---+---+
only showing top 3 rows

```

Figura 6.15_Spark Streaming: Ejecución Ejemplo 3 Cluster HDFS

6. Spark UI

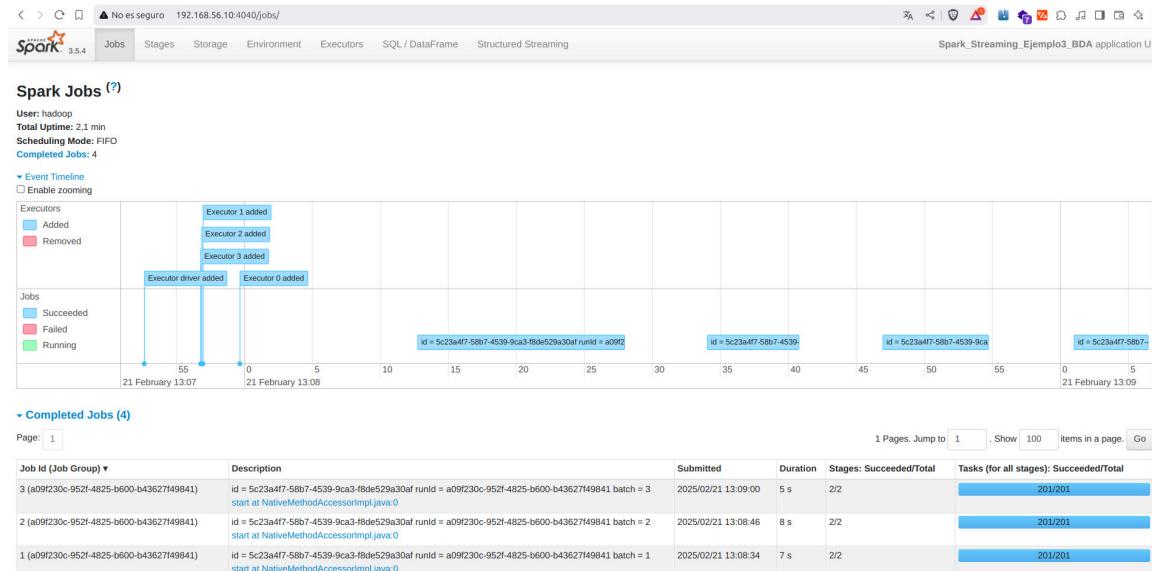


Figura 6.16_Spark Streaming: Ejecución Ejemplo 3 Cluster HDFS Spark UI

Plataformas a usar

Como ya hemos visto todas las opciones de solución en las diferentes plataformas, y sabemos usarlas, para los siguientes ejemplos se usarán **sólo una** de estas soluciones.

7. Window Operations

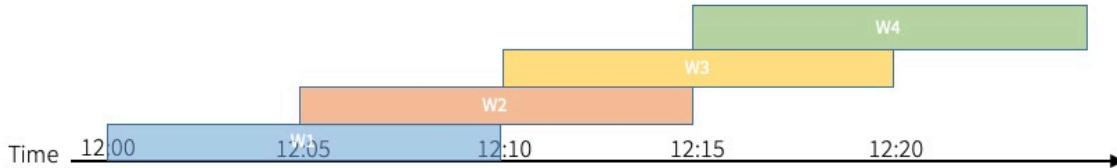
Como hemos visto en el punto 2.3 en la sección "*Handling Event-time and Late Data*", las operaciones de ventana (window operations) en Spark Streaming permiten agrupar datos en ventanas de tiempo para realizar cálculos agregados. Estas operaciones son esenciales para el procesamiento de flujos de datos en tiempo real, especialmente cuando se necesita analizar datos en intervalos específicos.

Hay tres tipos principales de operaciones de ventana en Spark Streaming: Tumbling (fixed) Windows, Sliding Windows y Session Windows. Cada uno tiene un propósito y uso específico dependiendo del tipo de análisis temporal que necesites realizar.

Tumbling Windows (5 mins)



Sliding Windows (10 mins, slide 5 mins)



Session Windows (gap duration 5 mins)



Figura 6.17_Spark Streaming: Window types (Fuente: spark.apache.org)

- **Tamaño fijo (tumbling/fixed window):** son ventanas de tiempo fijas y discretas que no se superponen entre sí. Cada elemento del stream solo puede pertenecer a una ventana. Estas ventanas avanzan en el tiempo de forma secuencial sin solaparse. Son útiles para

casos de uso donde quieras realizar cálculos agregados en intervalos de tiempo regulares y no necesitas solapamientos entre los intervalos.

- **Deslizantes (sliding window)**: permiten que las ventanas de tiempo se solapen. Estas ventanas tienen dos parámetros principales: el tamaño de la ventana (window length) y el desplazamiento de la ventana (slide interval). El tamaño de la ventana determina cuánto tiempo abarca cada ventana, mientras que el desplazamiento de la ventana especifica con qué frecuencia se inicia una nueva ventana. Esto significa que los elementos pueden pertenecer a múltiples ventanas, lo que es útil para análisis más finos y detallados que requieren superposiciones
- **De sesión (session windows)**: son dinámicas y se adaptan al comportamiento de los datos, agrupándolos en ventanas según la actividad del usuario o eventos específicos. Estas ventanas no tienen un tamaño fijo; en cambio, se definen por períodos de inactividad o gaps. Si no ocurren eventos nuevos dentro de un período de tiempo especificado (el gap de inactividad), la ventana se cierra, y cualquier evento nuevo iniciará una nueva ventana.

Para entender como funciona, vamos a basarnos en el ejemplo 1.

Modificaremos este programa para entender *windowing*. En lugar de realizar recuentos de palabras, queremos **contar palabras en períodos de 10 minutos y actualizarlas cada 5 minutos**. Es decir, contamos las palabras recibidas entre ventanas de 10 minutos, 12:00 - 12:10, 12:05 - 12:15, 12:10 - 12:20, etc. Ten en cuenta que 12:00 - 12:10 significa datos que llegaron después de las 12:00 pero antes de las 12:10.

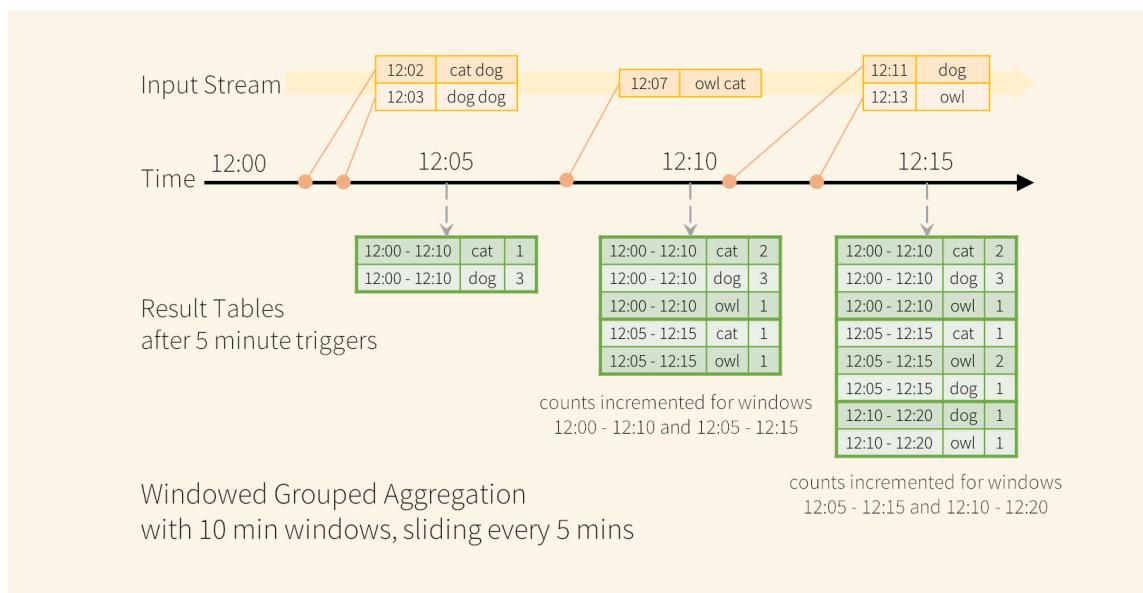


Figura 6.18_Spark Streaming: structured Streaming Window (Fuente: spark.apache.org)

Ahora, usando sliding window con intervalos cada 5 minutos, podemos considerar que una palabra que se recibió a las 12:07 incrementará los conteos correspondientes a dos ventanas: 12:00 - 12:10 y 12:05 - 12:15. Por lo tanto, los recuentos serán indexados tanto por la clave de

agrupación (es decir, la palabra) como por la ventana (se puede calcular a partir de la hora del evento).

A continuación, haremos 2 modificaciones en el ejemplo 1 para entender las diferencias entre fixed y sliding window

8. Ejemplo 4. Custom wordcount windowing

8.1 Fixed window

- Vamos a modificar el código para agrupar datos en una ventana fija de 1 minuto. Cuando leamos los datos, queremos obtener el **timestamp** de cada dato.

```

1  from pyspark.sql import SparkSession
2  from pyspark.sql.functions import explode
3  from pyspark.sql.functions import split
4  from pyspark.sql.functions import window
5
6  spark = SparkSession \
7      .builder \
8      .appName("CustomWordCount_BDA_fixed_window") \
9      .getOrCreate()
10
11 lines = spark \
12     .readStream \
13     .format("socket") \
14     .option("host", "localhost") \
15     .option("port", 9999) \
16     .option("includeTimestamp", True) \
17     .load()
```

- Obtenemos el texto escrito en el campo `value` y su `timestamp`:

```

1  lines.printSchema()
2  # root
3  # |-- value: string (nullable = true)
4  # |-- timestamp: timestamp (nullable = true)
```

- Preparamos el Dataframe para recoger el texto escrito y us timestamp

```

1  # Split the lines into words
2  words = lines.select(
3      explode(
4          split(lines.value, " ")).alias("word"),
5      lines.timestamp
6  )
```

- Ahora agrupamos en ventanas fijas de 1 minuto. Para ello, hacemos uso de la función `window`

```

1 # Generate running word count
2 wordCounts = words.groupBy("word").count()
3
4 windowedCounts = words.groupBy(
5     window(words.timestamp, "1 minute"),
6     words.word
7 ).count().orderBy("window")

```

5. Configuramos la salida de los datos por consola

```

1 # Start running the query that prints the running counts to the console
2 query = windowedCounts \
3     .writeStream \
4         .outputMode("complete") \
5             .option('truncate', 'false')\
6                 .format("console") \
7                     .start()\ \
8                         .awaitTermination()

```

6. La aplicación completa sería:

```

1 from pyspark.sql import SparkSession
2 from pyspark.sql.functions import explode
3 from pyspark.sql.functions import split
4 from pyspark.sql.functions import window
5
6 spark = SparkSession \
7     .builder \
8         .appName("CustomWordCount_BDA_fixed_window") \
9             .getOrCreate()
10
11 lines = spark \
12     .readStream \
13         .format("socket") \
14             .option("host", "localhost") \
15                 .option("port", 9999) \
16                     .option("includeTimestamp", True) \
17                         .load()
18
19 spark.sparkContext.setLogLevel("ERROR")
20
21 # Split the lines into words
22 words = lines.select(
23     explode(
24         split(lines.value, " ")).alias("word"),
25         lines.timestamp
26 )
27
28 # Generate running word count
29 wordCounts = words.groupBy("word").count()
30
31 windowedCounts = words.groupBy(
32     window(words.timestamp, "1 minute"),
33     words.word

```

```

34     ).count().orderBy("window")
35
36     # Start running the query that prints the running counts to the console
37     query = windowedCounts \
38         .writeStream \
39         .outputMode("complete") \
40         .option('truncate', 'false')\
41         .format("console") \
42         .start()\ \
43         .awaitTermination()

```

7. Ejecutamos el comando *NetCat* en el puerto 9999 que es donde nuestra app estará escuchando

```
1 nc -lk 9999
```

8. Ejecutamos nuestra app de Spark Streaming

9. Añadimos al socket lo siguiente en el minuto 13:44:00-13:45:00

```

1 Hola
2 Estamos en el IES Gran Capitán
3 Probando ventana
4 Módulo BDA

```

10. Y tenemos

```

1 -----
2 Batch: 1
3 -----
4 +-----+-----+
5 |window|word|count|
6 +-----+-----+
7 |{2024-03-04 13:44:00, 2024-03-04 13:45:00}|Hola|1|
8 |{2024-03-04 13:45:00, 2024-03-04 13:46:00}|el|1|
9 |{2024-03-04 13:45:00, 2024-03-04 13:46:00}|Capitán|1|
10 |{2024-03-04 13:45:00, 2024-03-04 13:46:00}|en|1|
11 |{2024-03-04 13:45:00, 2024-03-04 13:46:00}|Estamos|1|
12 |{2024-03-04 13:45:00, 2024-03-04 13:46:00}|Gran|1|
13 |{2024-03-04 13:45:00, 2024-03-04 13:46:00}|IES|1|
14 +-----+-----+
15
16 -----
17 Batch: 2
18 -----
19 +-----+-----+
20 |window|word|count|
21 +-----+-----+
22 |{2024-03-04 13:44:00, 2024-03-04 13:45:00}|Hola|1|
23 |{2024-03-04 13:45:00, 2024-03-04 13:46:00}|Estamos|1|
24 |{2024-03-04 13:45:00, 2024-03-04 13:46:00}|Gran|1|
25 |{2024-03-04 13:45:00, 2024-03-04 13:46:00}|Probando|1|
26 |{2024-03-04 13:45:00, 2024-03-04 13:46:00}|ventana|1|

```

```

27 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|el      |1      |
28 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|Capitán |1      |
29 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|en      |1      |
30 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|IES     |1      |
31 +-----+-----+
32
33 -----
34 Batch: 3
35 -----
36 +-----+-----+
37 |window          |word    |count|
38 +-----+-----+
39 | {2024-03-04 13:44:00, 2024-03-04 13:45:00}|Hola   |1      |
40 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|Estamos|1      |
41 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|BDA    |1      |
42 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|Gran   |1      |
43 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|Probando|1      |
44 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|ventana|1      |
45 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|IES    |1      |
46 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|el     |1      |
47 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|Módulo|1      |
48 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|Capitán|1      |
49 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|en     |1      |
50 +-----+-----+

```

11. En el siguiente minuto añadimos Probando ventana y obtenemos:

```

1 -----
2 Batch: 4
3 -----
4 +-----+-----+
5 |window          |word    |count|
6 +-----+-----+
7 | {2024-03-04 13:44:00, 2024-03-04 13:45:00}|Hola   |1      |
8 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|el     |1      |
9 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|Módulo|1      |
10 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|Capitán|1      |
11 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|en     |1      |
12 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|Estamos|1      |
13 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|BDA    |1      |
14 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|Gran   |1      |
15 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|Probando|1      |
16 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|IES    |1      |
17 | {2024-03-04 13:45:00, 2024-03-04 13:46:00}|ventana|1      |
18 | {2024-03-04 13:46:00, 2024-03-04 13:47:00}|Probando|1      |
19 | {2024-03-04 13:46:00, 2024-03-04 13:47:00}|ventana|1      |
20 +-----+-----+

```

```

hadoop@master:/opt/hadoop-3.3.0/spark-3.5.0/conf 104x48
2024-03-04 13:45:00, 2024-03-04 13:45:00){Probando 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){Ventana 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){el 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){Capitán 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){en 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){IES 1
...
Batch: 3
+-----+
|window |word |count|
+-----+
(2024-03-04 13:44:00, 2024-03-04 13:45:00){Hola 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){Estamos 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){BDA 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){Gran 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){Probando 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){ventana 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){IES 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){el 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){Modulo 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){Capitán 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){en 1
...
Batch: 4
+-----+
|window |word |count|
+-----+
(2024-03-04 13:44:00, 2024-03-04 13:45:00){Hola 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){Estamos 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){Modulo 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){Capitán 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){en 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){Estamos 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){BDA 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){Gran 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){Probando 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){IES 1
(2024-03-04 13:45:00, 2024-03-04 13:46:00){ventana 1
(2024-03-04 13:46:00, 2024-03-04 13:47:00){Probando 1
(2024-03-04 13:46:00, 2024-03-04 13:47:00){ventana 1
...

```

hadoop@master:/opt/spark/ejemplos/streaming/ejemplo4\$ nc -l 9999

Estamos en el TES Gran Capitán
Probando ventana
Modulo BDA
Probando ventana

Figura 6.19_Spark Streaming: Ejemplo 4. Fixed window

8.2 Sliding window

1. Cambiamos la siguiente parte del código para que mantenga la ventana de 1 minuto pero ahora tenga un deslizamiento de 30 segundos.

```

1 # Generate running word count
2 wordCounts = words.groupBy("word").count()
3
4 windowedCounts = words.groupBy(
5     window(words.timestamp, "1 minute", "30 seconds"),
6     words.word
7 ).count().orderBy("window")

```

2. La aplicación completa sería:

```

1 from pyspark.sql import SparkSession
2 from pyspark.sql.functions import explode
3 from pyspark.sql.functions import split
4 from pyspark.sql.functions import window
5
6 spark = SparkSession \
7     .builder \
8     .appName("CustomWordCount_BDA_sliding_window") \
9     .getOrCreate()
10
11 lines = spark \
12     .readStream \
13     .format("socket") \
14     .option("host", "localhost") \
15     .option("port", 9999) \
16     .option("includeTimestamp", True) \

```

```

17     .load()
18
19     spark.sparkContext.setLogLevel("ERROR")
20
21     # Split the lines into words
22     words = lines.select(
23         explode(
24             split(lines.value, " ")).alias("word"),
25             lines.timestamp
26     )
27
28     # Generate running word count
29     wordCounts = words.groupBy("word").count()
30
31     windowedCounts = words.groupBy(
32         window(words.timestamp, "1 minute", "30 seconds"),
33             words.word
34     ).count().orderBy("window")
35
36     # Start running the query that prints the running counts to the console
37     query = windowedCounts \
38         .writeStream \
39         .outputMode("complete") \
40         .option('truncate', 'false')\
41         .format("console") \
42         .start()\ \
43         .awaitTermination()

```

3. Ejecutamos el comando `NetCat` en el puerto 9999 que es donde nuestra app estará escuchando

```
1 nc -lk 9999
```

4. Ejecutamos nuestra app de Spark Streaming

5. Añadimos al socket `Hola Gran Capitán` en el timestamp `14:02:10`. Esto hace que introduzca las 3 palabras en las 2 ventanas existentes

```

1 -----
2 Batch: 1
3 -----
4 +-----+-----+-----+
5 |window|          |word|  |count|
6 +-----+-----+-----+
7 |{2024-03-04 14:01:30, 2024-03-04 14:02:30}|Gran| 1 |
8 |{2024-03-04 14:01:30, 2024-03-04 14:02:30}|Hola| 1 |
9 |{2024-03-04 14:01:30, 2024-03-04 14:02:30}|Capitán| 1 |
10 |{2024-03-04 14:02:00, 2024-03-04 14:03:00}|Hola| 1 |
11 |{2024-03-04 14:02:00, 2024-03-04 14:03:00}|Capitán| 1 |
12 |{2024-03-04 14:02:00, 2024-03-04 14:03:00}|Gran| 1 |
13 +-----+-----+-----+

```

6. Si añadimos `Hola mundo`, el siguiente microbatch lo añade en las ventanas correspondientes. Observa que `hola` en la ventana `14:02:00-14:03:00` aparece 2 veces.

1	-----
2	Batch: 2
3	-----
4	+-----+-----+
5	window word count
6	+-----+-----+
7	{2024-03-04 14:01:30, 2024-03-04 14:02:30} Gran 1
8	{2024-03-04 14:01:30, 2024-03-04 14:02:30} Hola 1
9	{2024-03-04 14:01:30, 2024-03-04 14:02:30} Capitán 1
10	{2024-03-04 14:02:00, 2024-03-04 14:03:00} Hola 2
11	{2024-03-04 14:02:00, 2024-03-04 14:03:00} Capitán 1
12	{2024-03-04 14:02:00, 2024-03-04 14:03:00} mundo 1
13	{2024-03-04 14:02:00, 2024-03-04 14:03:00} Gran 1
14	{2024-03-04 14:02:30, 2024-03-04 14:03:30} mundo 1
15	{2024-03-04 14:02:30, 2024-03-04 14:03:30} Hola 1
16	+-----+-----+

```

hadoop@master:/opt/hadoop-3.3.6/spark-3.5.0/conf 104x48
24/03/04 14:02:07 INFO StandaloneSchedulerBackend$StandaloneDriverEndpoint: Registered executor NettyRpcEndpointRef(spark-client://Executor) (192.168.11.11:35764) with ID 1, ResourceProfileId 0
24/03/04 14:02:07 INFO StandaloneSchedulerBackend$StandaloneDriverEndpoint: Registered executor NettyRpcEndpointRef(spark-client://Executor) (192.168.11.13:35104) with ID 0, ResourceProfileId 0
24/03/04 14:02:07 INFO BlockManagerMasterEndpoint: Registering block manager 192.168.11.13:38539 with 36.63 MB RAM, BlockManagerId(0, 192.168.11.13, 38539, None)
24/03/04 14:02:07 INFO BlockManagerMasterEndpoint: Registering block manager 192.168.11.13:32775 with 36.63 MB RAM, BlockManagerId(0, 192.168.11.13, 32775, None)
Batch: 0
-----[window|word|count]-----+-----+
Batch: 1
-----[window|word|count]-----+-----+
| {2024-03-04 14:01:30, 2024-03-04 14:02:30} | Gran | 1 |
| {2024-03-04 14:01:30, 2024-03-04 14:02:30} | Hola | 1 |
| {2024-03-04 14:01:30, 2024-03-04 14:02:30} | Capitán | 1 |
| {2024-03-04 14:02:00, 2024-03-04 14:03:00} | Gran | 1 |
| {2024-03-04 14:02:00, 2024-03-04 14:03:00} | Capitán | 1 |
| {2024-03-04 14:02:00, 2024-03-04 14:03:00} | Gran | 1 |
Batch: 2
-----[window|word|count]-----+-----+
| {2024-03-04 14:01:30, 2024-03-04 14:02:30} | Gran | 1 |
| {2024-03-04 14:01:30, 2024-03-04 14:02:30} | Hola | 1 |
| {2024-03-04 14:01:30, 2024-03-04 14:02:30} | Capitán | 1 |
| {2024-03-04 14:02:00, 2024-03-04 14:03:00} | Hola | 2 |
| {2024-03-04 14:02:00, 2024-03-04 14:03:00} | Capitán | 1 |
| {2024-03-04 14:02:00, 2024-03-04 14:03:00} | mundo | 1 |
| {2024-03-04 14:02:30, 2024-03-04 14:03:30} | mundo | 1 |
| {2024-03-04 14:02:30, 2024-03-04 14:03:30} | Hola | 1 |

```

Figura 6.20_Spark Streaming: Ejemplo 4. Sliding window

9. Watermarking

En términos generales, cuando se trabaja con datos en streaming, habrá retrasos entre el tiempo del evento y el tiempo de procesamiento debido a cómo se ingieren los datos y si la aplicación en general experimenta problemas como tiempo de inactividad. Debido a estos posibles retrasos variables, el motor que utiliza para procesar estos datos debe tener algún mecanismo para decidir cuándo cerrar las ventanas agregadas y producir el resultado agregado.

Para explicar esto visualmente, vamos a usar 2 escenarios:

9.1 Escenario 1. Temperatura y presión

Vamos a tomar un escenario en el que recibimos datos en varios momentos, aproximadamente entre las 10:50 AM y las 11:20 AM. Estamos creando ventanas oscilantes de 10 minutos que calculan el promedio de las lecturas de temperatura y presión que llegaron durante el período de ventana.

En esta primera imagen, tenemos *fixed window* a las 11:00 AM, 11:10 AM y 11:20 AM, lo que lleva a las tablas de resultados que se muestran en los horarios respectivos. Cuando el segundo lote de datos llega alrededor de las 11:10 AM con datos que tienen una hora de evento de 10:53 AM, esto se incorpora a los promedios de temperatura y presión calculados para la ventana de 11:00 AM → 11:10 AM que se cierra a las 11:10 AM, lo que no da el resultado correcto.

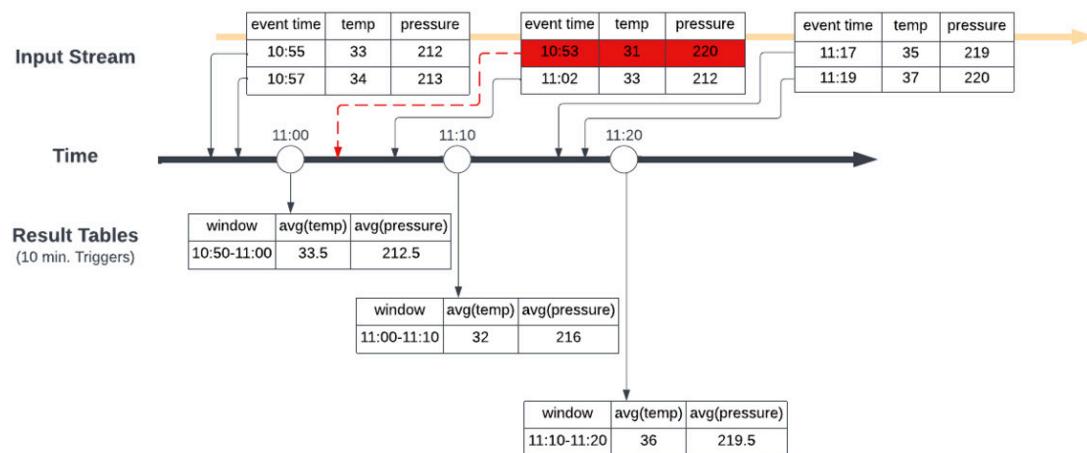


Figura 6.21_Spark Streaming: Watermarking 1 (Fuente: databricks.com)

Para asegurarnos obtener los resultados correctos para los agregados que queremos producir, debemos definir un **watermarking** (marca de agua) que le permitirá a Spark saber cuándo cerrar la ventana de agregados y producir el resultado agregado correcto.

En las aplicaciones de Structured Streaming, en el sentido más básico, al definir un **watermarking**, Spark Structured Streaming sabe cuándo ha ingerido todos los datos hasta cierto tiempo, T , (según una expectativa de retraso establecida) para que pueda cerrar y producir agregados en ventanas hasta la marca de tiempo T .

Esta segunda imagen muestra el efecto de implementar un **watermarking de 10 minutos**, un **triggers de 10 minutos** y usar el **append mode** en Spark Structured Streaming.

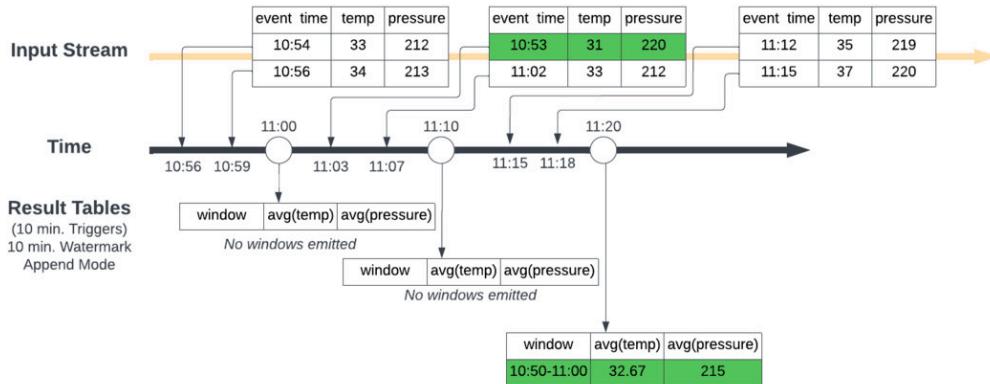


Figura 6.22_Spark Streaming: Watermarking 2 (Fuente: databricks.com)

A diferencia del primer escenario, Spark ahora espera para cerrar y generar *windowed aggregation* una vez que el **tiempo máximo del evento visto menos el watermarking especificado sea mayor que el límite superior de la ventana**.

En otras palabras, Spark necesitaba esperar hasta ver puntos de datos en los que la última hora del evento vista menos 10 minutos fuera mayor que las 11:00 AM para emitir la ventana agregada de 10:50 AM → 11:00 AM. A las 11:00 AM, no ve esto, por lo que solo inicializa el cálculo agregado en el almacén de estado interno de Spark. A las 11:10 AM, esta condición aún no se cumple, pero tenemos un nuevo punto de datos para las 10:53 AM, por lo que el estado interno se actualiza, pero no se emite. Luego, finalmente, a las 11:20 AM, Spark vio un punto de datos con una hora de evento de 11:15 AM y desde las 11:15 AM menos 10 minutos son las 11:05 AM, que es más tarde del 11:00 AM del_windowing_ 10:50 AM → 11:00 AM, por tanto se puede emitir una ventana de las 11:00 AM a la tabla de resultados.

Esto produce el resultado correcto al incorporar adecuadamente los datos en función del retraso esperado definido por la marca de agua. Una vez que se emiten los resultados, el estado correspondiente se elimina del almacén de estados.

9.2 Escenario 2. Word Count

Vamos a tomar como escenario 2, usaremos de nuevo como ejemplo la aplicación `word count`.

Por ejemplo, la aplicación podría recibir una palabra generada a las 12:04 (es decir, la hora del evento) a las 12:11. La aplicación debe usar la hora 12:04 en lugar de 12:11 para actualizar los recuentos anteriores para la ventana de 12:00 a 12:10. Esto ocurre naturalmente en nuestra *window-based grouping*: Structured Streaming puede mantener el estado intermedio para agregados parciales durante un largo período de tiempo, de modo que los datos tardíos puedan actualizar correctamente los agregados de ventanas antiguas, como se ilustra a continuación.

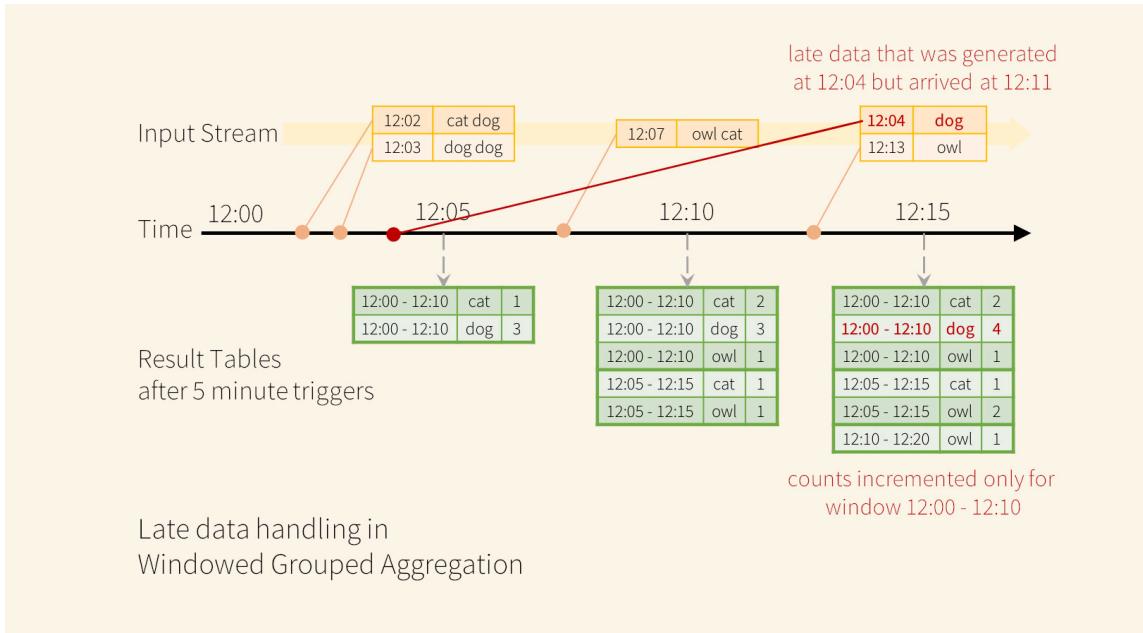


Figura 6.22_Spark Streaming: Watermarking 3 (Fuente: spark.apache.org)

Sin embargo, como ya hemos visto, para ejecutar esta consulta durante días, es necesario que el sistema limite la cantidad de estado intermedio en memoria que acumula. Esto significa que el sistema necesita saber cuándo se puede eliminar un agregado antiguo del estado en memoria porque la aplicación ya no recibirá datos tardíos para ese agregado. **Watermarking** permite que el motor rastree automáticamente la hora del evento actual en los datos e intente limpiar el estado anterior en consecuencia. Vamos a usar el ejemplo para entenderlo

Podemos definir fácilmente las marcas de agua en el ejemplo anterior usando `withWatermark()` como se muestra a continuación.

```

1 words = ... # streaming DataFrame of schema { timestamp: Timestamp, word: String }
2
3 # Group the data by window and word and compute the count of each group
4 windowedCounts = words \
5     .withWatermark("timestamp", "10 minutes") \
6     .groupByKey( \
7         window(words.timestamp, "10 minutes", "5 minutes"), \
8         words.word) \
9     .count()

```

Update Mode

En este ejemplo, definimos la **watermark** de la consulta en el valor de la columna "timestamp" y también definimos "10 minutos" como el umbral de qué tan tarde se permite que lleguen los datos. Si esta consulta se ejecuta en el *update mode* de salida el motor seguirá actualizando los recuentos de una *window* en la tabla de resultados hasta que la ventana sea más antigua

que la watermark, que va por detrás del tiempo del evento actual en la columna "timestamp" por 10 minutos.

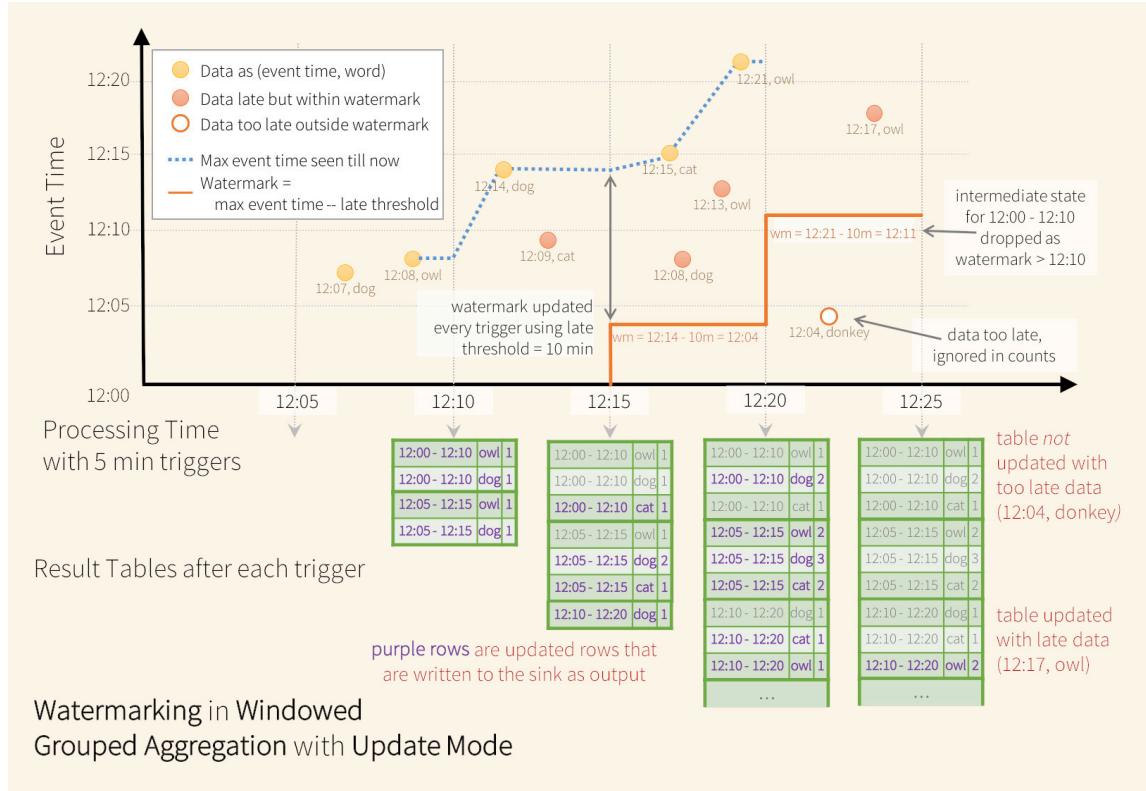


Figura 6.24_Spark Streaming: Watermarking 4 (Fuente: spark.apache.org)

Como se muestra en la figura, el tiempo máximo del evento rastreado por el motor es la línea discontinua azul y la watermark establecida ($\text{max event time} - 10 \text{ mins}$) al comienzo de cada trigger es la línea roja.

Append Mode

Por ejemplo, cuando el motor observa los datos (12:14, dog), establece la watermark para el siguiente trigger en 12:04. Esta marca de agua permite que el motor mantenga un estado intermedio durante 10 minutos adicionales para permitir que se cuenten los datos tardíos. Por ejemplo, los datos (12:09, cat) están desordenados y retrasados, y caen en las ventanas 12:00 - 12:10 y 12:05 - 12:15. Dado que todavía está por delante de la marca de agua 12:04 en el trigger, el motor aún mantiene los recuentos intermedios como estado y actualiza correctamente los recuentos de las ventanas relacionadas. Sin embargo, cuando la marca de agua se actualiza a las 12:11, el estado intermedio de la ventana (12:00 - 12:10) se borra y todos los datos posteriores (por ejemplo, (12:04, donkey)) se consideran "demasiado tarde (too late)" y por lo tanto ignorado. Tengamos en cuenta que después de cada trigger, los recuentos actualizados (es decir, filas moradas) se escriben para el sink de salida configurado, según lo dicta el update mode.

Es posible que algunos receptores(sink) (por ejemplo, *files*) no admitan las actualizaciones detalladas que requiere el update mode. Para trabajar con ellos, también admitimos append mode, donde solo se escriben los recuentos finales para *sink* de salida. Lo vemos en la siguiente figura.

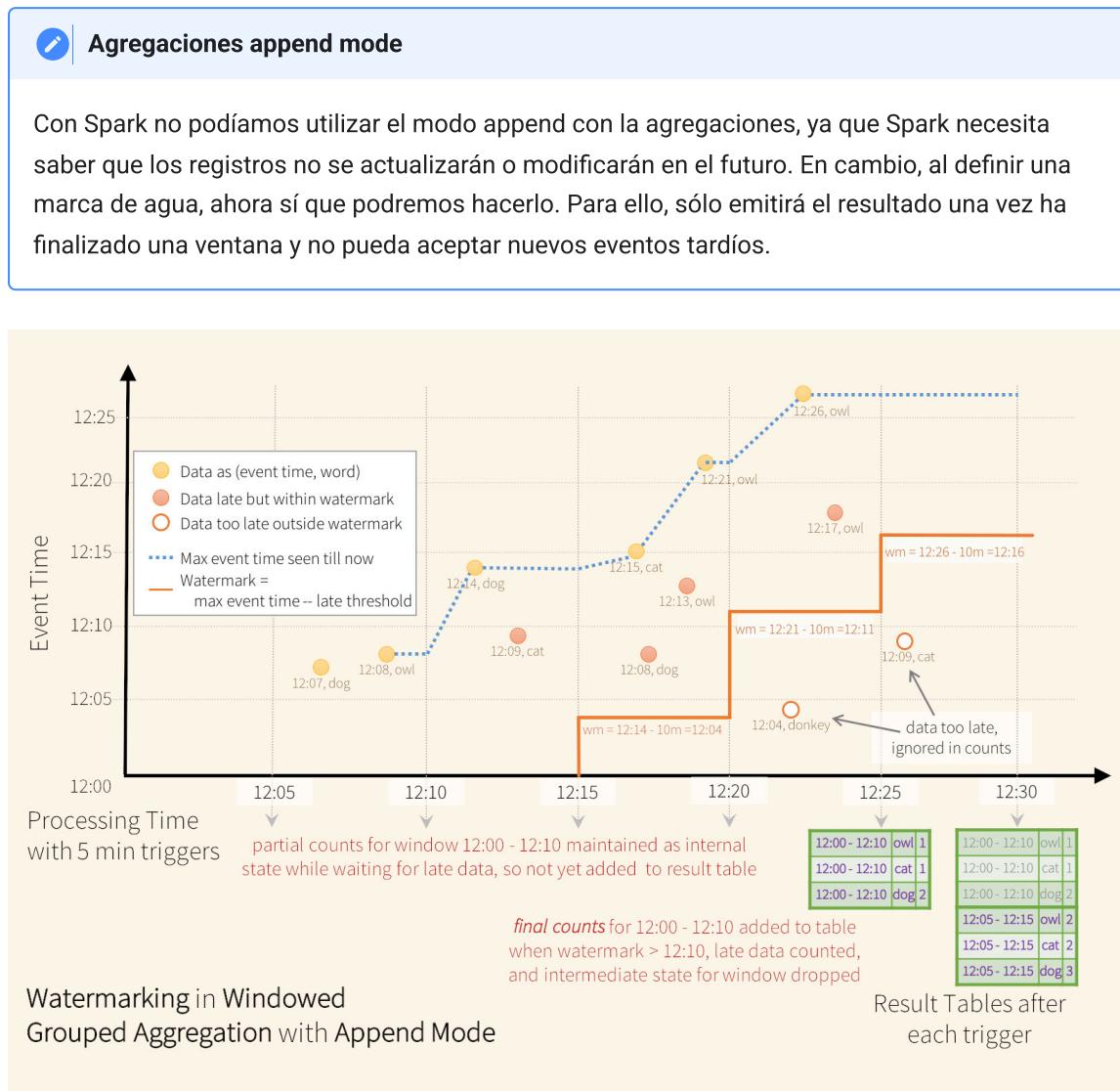


Figura 6.25_Spark Streaming: Watermarking 5 (Fuente: spark.apache.org)

Tengamos en cuenta que el uso de `withWatermark` en un `non-streaming Dataset` no es operativo. Como la watermark no debería afectar a ninguna consulta por lotes de ninguna manera, se ignorará directamente.

10. Ejemplo 5. Window y Watermark

Vamos a realizar un ejercicio para entender el funcionamiento de **watermark**

Para ello usaremos el siguiente código, donde leemos a través de un socket en el puerto 9999 un registro que contiene un event_time(timestamp) con una # y un número que nos servirá de sumatorio para ir controlando a qué ventana se procesan los datos. El registro tendrá el siguiente contenido 2024-03-11 10:06:00#2

1. Código de la aplicación

ejemplo5_watermark.py

```

1  from pyspark.sql import SparkSession
2  from pyspark.sql.functions import *
3
4  spark = SparkSession \
5      .builder \
6      .appName("Ejemplo5_watermark") \
7      .getOrCreate()
8
9  df_init = spark \
10     .readStream \
11     .format("socket") \
12     .option("host", "localhost") \
13     .option("port", 9999) \
14     .load()
15
16 # Set Spark logging level to ERROR to avoid various other logs on
17 # console.
17 spark.sparkContext.setLogLevel("ERROR")
18
19 df_event = df_init.select(split(col("value"), "#").alias("data")) \
20     .withColumn("event_timestamp", element_at(col("data"),
21 1).cast("timestamp")) \
22     .withColumn("val", element_at(col("data"), 2).cast("integer")) \
23     .drop("data")
24
25 df_result = df_event \
26     .withWatermark("event_timestamp", "10 minutes") \
27     .groupBy(window(col("event_timestamp"), "5 minutes")) \
28     .agg(sum("val").alias("sum"))
29
30 query = df_result \
31     .writeStream \
32     .outputMode("update") \
33     .option('truncate', 'false') \
34     .format("console") \
35     .start() \
36     .awaitTermination()
```

2. Hemos configurado una **ventana de 5 minutos** con una **marca de agua de 10 minutos**. Esto significa que Spark espera hasta 10 minutos **después del último timestamp más reciente para considerar los datos para la ventana correspondiente**, permitiendo cierta flexibilidad para el procesamiento de datos que llegan tarde.

3. Ejecutamos el comando `NetCat` en el puerto 9999 que es donde nuestra app estará escuchando

```
1 | nc -lk 9999
```

4. Ejecutamos nuestra app de Spark Streaming

```
1 | spark-submit --master spark://192.168.11.10:7077 /opt/hadoop-  
3.4.1/spark/ejemplos/streaming/ejemplo5/ejemplo5_watermark.py
```

5. Primer registro: `2024-03-11 10:06:00#2`. El registro se procesa dentro de su `window` correspondiente de 5 minutos

```
1 | -----  
2 | Batch: 1  
3 | -----  
4 |-----+---+  
5 | window | sum |  
6 |-----+---+  
7 | {2024-03-11 10:05:00, 2024-03-11 10:10:00} |2 |  
8 |-----+---+
```

6. Segundo registro: `2024-03-11 10:13:00#5`.

- a. Hasta ahora sólo hemos procesado un registro, lo que significa `max event time = 2024-03-11 10:06:00` (es decir, el último timestamp más reciente, *recuerda el punto 2*). Por lo tanto, la marca de agua `max event time - delayThreshold = 2024-03-11 09:56:00`. Para una ventana de tamaño 5 minutos, la ventana de la marca de agua será `{09:55:00, 10:00:00}`.
- b. Por tanto, cualquier dato anterior al inicio de la ventana de la marca de agua `{09:55:00, 10:00:00}` serían descartados.
- c. Sin embargo, el tiempo del evento `{2024-03-11 10:13:00}` para el segundo registro es mayor que la hora de inicio de la ventana de la marca de agua, por lo que se procesará.

```
1 | -----  
2 | Batch: 3  
3 | -----  
4 |-----+---+  
5 | window | sum |  
6 |-----+---+  
7 | {2024-03-11 10:10:00, 2024-03-11 10:15:00} |5 |  
8 |-----+---+
```

7. Tercer registro: `2024-03-10 10:06:00#1`. El estado actual del sistema es:

- a. **Tiempo máximo del evento:** `{2024-03-11 10:13:00}`

b. **watermark:** {2024-03-11 10:03:00}

c. **window watermark:** {2024-03-11 10:00:00, 2024-03-11 10:05:00}

d. El tiempo es menor que el inicio de la ventana de la marca de agua, por lo tanto, se tratará como datos tardíos y se descartará. Obtendremos un minibatch vacío.

```

1 -----
2 Batch: 5
3 -----
4 +---+---+
5 | window | sum |
6 +---+---+
7 +---+---+

```

7. Cuarto registro: 2024-03-11 10:08:00#8 . El estado actual del sistema es:

a. **Tiempo máximo del evento:** {2024-03-11 10:13:00}

b. **watermark:** {2024-03-11 10:03:00}

c. **window watermark:** {2024-03-11 10:00:00, 2024-03-11 10:05:00}

d. Este tiempo de eventos es mayor que el inicio de la ventana de la marca de agua, por lo tanto, se procesará (con su suma correspondiente del valor del `group by` de la ventana).

```

1 -----
2 Batch: 6
3 -----
4 +-----+---+
5 | window | sum |
6 +-----+---+
7 | {2024-03-11 10:05:00, 2024-03-11 10:10:00} | 10 |
8 +-----+---+

```

8. Quinto registro: 2024-03-11 10:00:00#7 . El estado actual del sistema es:

a. **Tiempo máximo del evento:** {2024-03-11 10:13:00}

b. **watermark:** {2024-03-11 10:03:00}

c. **window watermark:** {2024-03-11 10:00:00, 2024-03-11 10:05:00}

d. Este tiempo de evento **es igual que el inicio de la ventana de la marca de agua**, por lo tanto, se procesará.

```

1 -----
2 Batch: 7
3 -----
4 +-----+---+
5 | window | sum |
6 +-----+---+
7 | {2024-03-11 10:00:00, 2024-03-11 10:05:00} | 7 |

```

8

-----+---+

9. Sexto registro: 2024-03-11 9:59:00#3 . El estado actual del sistema es:

- a. **Tiempo máximo del evento:** {2024-03-11 10:13:00}
- b. **watermark:** {2024-03-11 10:03:00}
- c. **window watermark:** {2024-03-11 10:00:00, 2024-03-11 10:05:00}
- d. Este tiempo de evento **es menor que el inicio de la ventana de la marca de agua**, por lo tanto, no se procesará.

1

2

Batch: 8

3

-----+---+

4

| window | sum |

5

+-----+---+

6

+-----+---+

7

+-----+---+

10. Séptimo registro: 2024-03-11 10:22:00#5 . El estado actual del sistema es:

- a. **Tiempo máximo del evento:** {2024-03-11 10:13:00}
- b. **watermark:** {2024-03-11 10:03:00}
- c. **window watermark:** {2024-03-11 10:00:00, 2024-03-11 10:05:00}
- d. Este tiempo de eventos es mayor que el inicio de la ventana de la marca de agua, por lo tanto, se procesará

1

2

Batch: 9

3

-----+---+

4

| window | sum |

5

+-----+---+

6

+-----+---+

7

| {2024-03-11 10:20:00, 2024-03-11 10:25:00} | 5 |

8

+-----+---+

11. Octavo registro: 2024-03-11 10:09:00#4 . El estado actual del sistema es:

- a. **Tiempo máximo del evento:** {2024-03-11 10:22:00}
- b. **watermark:** {2024-03-11 10:12:00}
- c. **window watermark:** {2024-03-11 10:10:00, 2024-03-11 10:15:00}
- d. Este tiempo de evento **es menor que el inicio de la ventana de la marca de agua**, por lo tanto, no se procesará.

1

2

Batch: 11

```

3 -----
4 +----+---+
5 |window|sum|
6 +----+---+
7 +----+---+

```

12. Y así sucesivamente con otros eventos

Evento	Tiempo máximo del evento	watermark	window watermark	Procesado
2024-03-11 10:06:00#2				
2024-03-11 10:13:00#5	{2024-03-11 10:06:00}	{2024-03-11 09:56:00}	{2024-03-11 09:55:00, 2024- 03-11 10:00:00}	Si
2024-03-10 10:06:00#1	{2024-03-11 10:13:00}	{2024-03-11 10:03:00}	{2024-03-11 10:00:00, 2024- 03-11 10:05:00}	No
2024-03-11 10:08:00#8	{2024-03-11 10:13:00}	{2024-03-11 10:03:00}	{2024-03-11 10:00:00, 2024- 03-11 10:05:00}	Si
2024-03-11 10:00:00#7	{2024-03-11 10:13:00}	{2024-03-11 10:03:00}	{2024-03-11 10:00:00, 2024- 03-11 10:05:00}	Si
2024-03-11 10:22:00#5	{2024-03-11 10:13:00}	{2024-03-11 10:03:00}	{2024-03-11 10:00:00, 2024- 03-11 10:05:00}	Si
2024-03-11 10:09:00#4	{2024-03-11 10:22:00}	{2024-03-11 10:12:00}	{2024-03-11 10:10:00, 2024- 03-11 10:15:00}	No
2024-03-11 10:11:00#6	{2024-03-11 10:22:00}	{2024-03-11 10:12:00}	{2024-03-11 10:10:00, 2024- 03-11 10:15:00}	Si

Evento	Tiempo máximo del evento	watermark	window watermark	Procesado
2024-03-11 10:35:00#2	{2024-03-11 10:22:00}	{2024-03-11 10:12:00}	{2024-03-11 10:10:00, 2024-03-11 10:15:00}	Si
2024-03-11 10:24:00#9	{2024-03-11 10:35:00}	{2024-03-11 10:25:00}	{2024-03-11 10:25:00, 2024-03-11 10:30:00}	No
2024-03-11 10:25:00#1	{2024-03-11 10:35:00}	{2024-03-11 10:25:00}	{2024-03-11 10:25:00, 2024-03-11 10:30:00}	Si

Tabla 6.1: Ejemplo de watermark

Dada la propia idiosincrasia de Spark streaming, se generan algunos microbatch vacíos sin introducción de registros

```
[root@hadoop master]# hadoop@master:/opt/spark/ejemplos/streaming/ejemplo5$ nc -l 9999
2024-03-11 10:06:00#2
2024-03-11 10:13:00#5
2024-03-11 10:20:00#1
2024-03-11 10:28:00#8
2024-03-11 10:35:00#7
2024-03-11 9:59:00#3
2024-03-11 10:22:00#5
2024-03-11 10:09:00#4
2024-03-11 10:11:00#6
2024-03-11 10:18:00#2
2024-03-11 10:24:00#9
2024-03-11 10:25:00#1

[|window|sum]
+-----+-----+
+-----+-----+
+-----+-----+
Batch: 12
+-----+-----+
+-----+-----+
+-----+-----+
[|window|sum]
+-----+-----+
+-----+-----+
+-----+-----+
[(2024-03-11 10:10:00, 2024-03-11 10:15:00)|11|]
+-----+-----+
+-----+-----+
+-----+-----+
Batch: 13
+-----+-----+
+-----+-----+
+-----+-----+
[|window|sum]
+-----+-----+
+-----+-----+
+-----+-----+
[(2024-03-11 10:35:00, 2024-03-11 10:40:00)|2|]
+-----+-----+
+-----+-----+
+-----+-----+
Batch: 14
+-----+-----+
+-----+-----+
+-----+-----+
[|window|sum]
+-----+-----+
+-----+-----+
+-----+-----+
+-----+-----+
+-----+-----+
Batch: 15
+-----+-----+
+-----+-----+
+-----+-----+
[|window|sum]
+-----+-----+
+-----+-----+
+-----+-----+
+-----+-----+
+-----+-----+
Batch: 16
+-----+-----+
+-----+-----+
+-----+-----+
[|window|sum]
+-----+-----+
+-----+-----+
+-----+-----+
[(2024-03-11 10:25:00, 2024-03-11 10:30:00)|1|]
+-----+-----+
+-----+-----+
+-----+-----+
```

Figura 6.26_Spark Streaming: Ejemplo Watermark