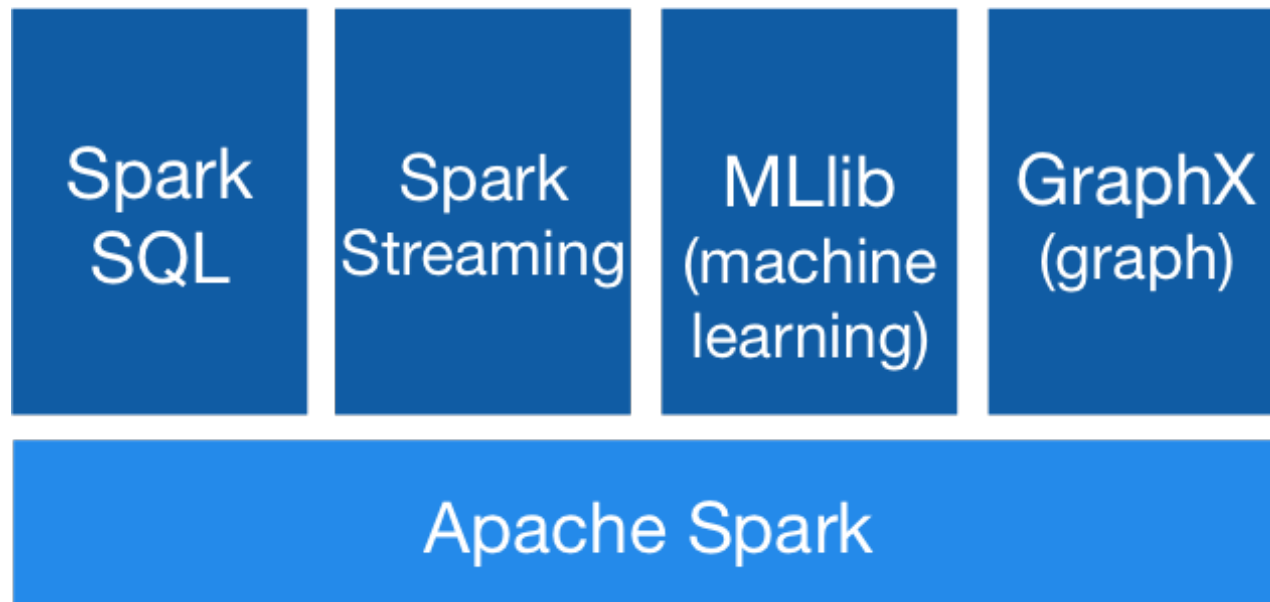




Spark Streaming

CIFF

Spark Streaming - Contexto



Spark Streaming – Contexto

Batch

High throughput

Interactive

Low latency

Stream

Continuos

Iterative

processing

Algorithm processing

Spark Streaming

“Spark Streaming es una extensión del core de Spark que permite procesamiento escalable, con un alto throughput y tolerante a fallos de streams de datos en tiempo real”

Spark Streaming - Características

- **Operadores de alto nivel.** El API de Spark Streaming permite utilizar una librería Java, Scala y Python que permite realizar procesamiento Streaming como si fuera Batch.
- **Tolerancia a fallos.** Nos permite una semántica de exactamente una vez. El procesamiento puede ser recuperado desde la última vez que fallo.
- **Integración con Spark.** Se puede combinar análisis en batch, streaming y queries interactivas.
- **Múltiples fuentes y destinos de datos.**
- **Desplegable en distintos formatos:** standalone, cluster o en la nube.

Spark Streaming - Funcionamiento

- 1.El data es recogido desde una fuente:
 - Kafka
 - Flume
 - Twitter
 - ZeroMQ
 - TCP Sockets...
- 2.Procesamiento del dato utilizando funciones Spark de alto nivel como map, reduce, window, ...
- 3.El dato procesado es enviado a fichero, base de datos, ...

Spark Streaming - Funcionamiento



Spark Streaming – Primeros pasos

- Hola, Mundo! Aka Wordcount
- Pasos:
 - Arrancar un contexto Spark en Streaming, `StreamingContext`.
 - Abrir sobre dicho contexto un socket para leer un stream, `socketTextStream`
 - Hacer exactamente el mismo map y reduce para WordCount como si estuviéramos en el core de Spark.
 - Comenzar a escuchar por el socket stream y esperar a que se finalice el procesamiento (o indicar el tiempo hasta que este termine)

Spark Streaming - Ejemplo

```
object NetworkWordCount {  
  def main(args: Array[String]) {  
    if (args.length < 2) {  
      System.err.println("Usage: NetworkWordCount <hostname> <port>")  
      System.exit(1)  
    }  
  
    StreamingExamples.setStreamingLogLevels()  
  
    // Create the context with a 1 second batch size  
    val sparkConf = new SparkConf().setAppName("NetworkWordCount")  
    val ssc = new StreamingContext(sparkConf, Seconds(1))  
  
    // Create a socket stream on target ip:port and count the  
    // words in input stream of \n delimited text (eg. generated by 'nc')  
    // Note that no duplication in storage level only for running locally.  
    // Replication necessary in distributed scenario for fault tolerance.  
    val lines = ssc.socketTextStream(args(0), args(1).toInt, StorageLevel.MEMORY_AND_DISK_SER)  
    val words = lines.flatMap(_.split(" "))  
    val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)  
    wordCounts.print()  
    ssc.start()  
    ssc.awaitTermination()  
  }  
}
```

Spark Streaming - Programación

- Para programar en Spark Streaming, al igual que se hace con Spark Core se tiene que añadir una dependencia al proyecto.

- Con Maven

```
<dependency>
```

```
  <groupId>org.apache.spark</groupId>
```

```
  <artifactId>spark-streaming_2.10</artifactId>
```

```
  <version>1.5.2</version>
```

```
</dependency>
```

- Con SBT

```
libraryDependencies += "org.apache.spark" %  
"spark-streaming_2.10" % "1.5.2"
```

Spark Streaming - Programación

Source

Kafka

Flume

Kinesis

Twitter

ZeroMQ

MQTT

Artifact

spark-streaming-kafka_2.10

spark-streaming-flume_2.10

spark-streaming-kinesis-asl_2.10
[Amazon Software License]

spark-streaming-twitter_2.10

spark-streaming-zeromq_2.10

spark-streaming-mqtt_2.10

Spark Streaming – Streaming Context

- El punto de entrada para toda aplicación de Spark Streaming es nuestro Streaming Context
- Dicho Streaming Context puede ser creado de dos maneras diferentes:
 - A partir de un objeto **SparkConf**
 - A partir de un objeto **Spark Context**
- Ambas formas de crear el Streaming Context ofrecen la misma funcionalidad

Spark Streaming - SparkConf

- Define la configuración para una aplicación de Spark
- La configuración es definida mediante parametros de pares clave-valor
- Si se llama al constructor vacío `SparkConf()` se cargarán aquellas propiedades de sistema Java que sean del tipo `spark.*`
- Al crear el Streaming Context con un objeto `SparkConf` internamente se creará un contexto de Spark.

Spark Streaming - SparkConf

- Uso:

```
import org.apache.spark._  
import org.apache.spark.streaming._  
  
val conf = new SparkConf().setAppName(appName).setMaster(master)  
val ssc = new StreamingContext(conf, Seconds(1))
```

Spark Streaming – Spark Context

- Breve recordatorio:
 - El Contexto de Spark representa la conexión entre la aplicación y el cluster de Spark
 - Permite ser utilizado para crear RDDs, acumuladores, variables broadcast ... y un streaming context.
 - Sólo va a existir un Spark Context por JVM

Spark Streaming – Spark Context

- Uso:

```
import org.apache.spark.streaming._  
  
val sc = ...           // existing SparkContext  
val ssc = new StreamingContext(sc, Seconds(1))
```


Spark Streaming – Recordar

I

- Después de que el contexto haya sido definido:
 - Definir la entrada a partir de **Dstreams**.
 - Definir **operaciones de transformación y salida** sobre los Dstreams.
 - Comenzar a recibir datos utilizando el método **streamingContext.start()**.
 - Esperar que el procesamiento en streaming se **pare** manualmente mediante el método `streamingContext.stop()` o debido a un error.

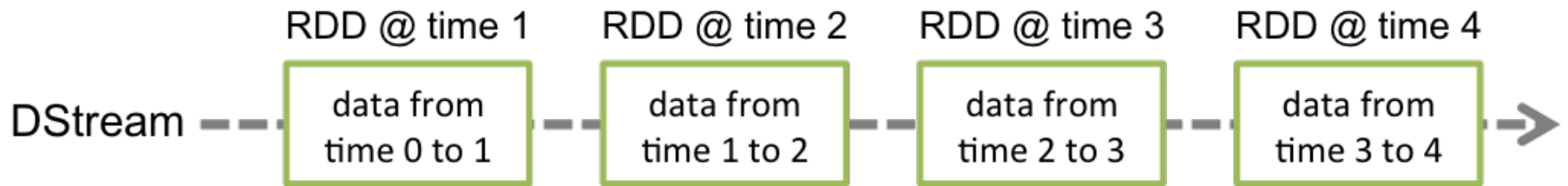
Spark Streaming – Recordar II

- Una vez que un contexto ha comenzado, no se puede añadir más computo en streaming.
- Una vez que un contexto se ha parado, no puede ser reiniciado
- Sólo puede existir un streaming context en una JVM cada vez
- El método `stop()` para tanto el streaming context como el Spark Context. Para parar únicamente el streaming context hay que setear el parametro 'stopSparkContext' a falso.
- El SparkContext puede ser reutilizado para crear multiples streaming context, en tanto que no paremos el SparkContext también.

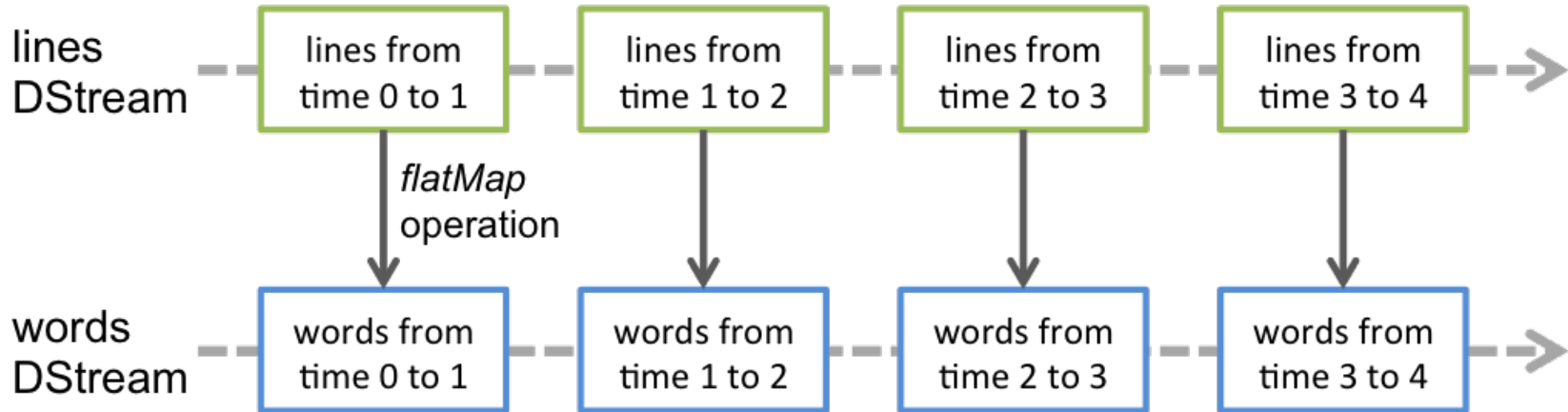
Spark Streaming - DStreams

- **Discretized Stream** o **Dstream** es la abstracción básica que ofrece Spark para procesamiento en streaming.
- Representa un stream de data continuo, a partir de una fuente o de otro stream ya procesado
- Un Dstream está compuesto por una serie de RDDs (**microbatching**) que contiene datos cada cierto intervalo

Spark Streaming - DStreams



Spark Streaming - DStreams

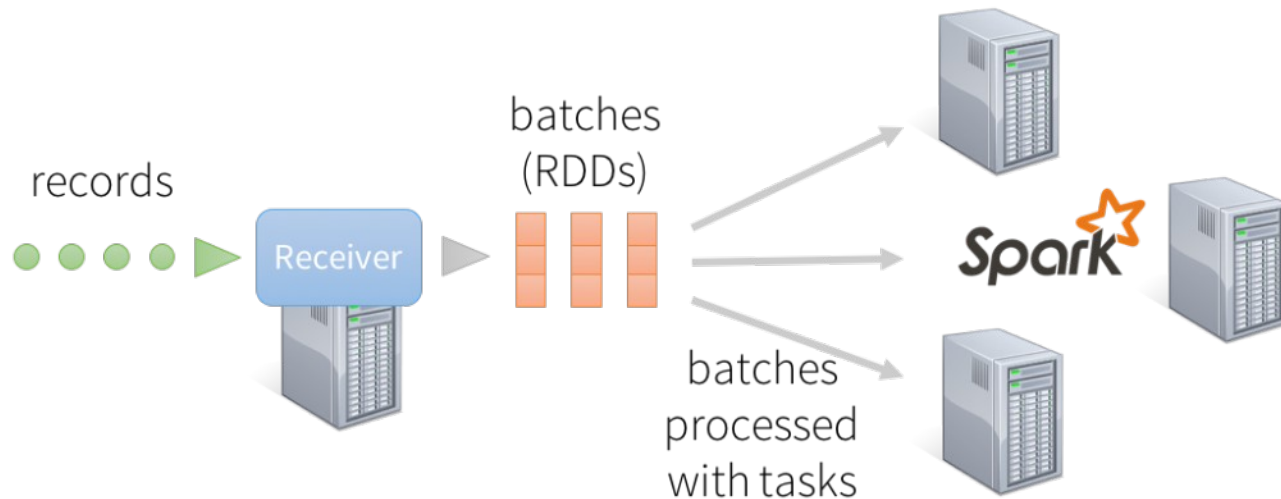


Spark Streaming – Input DStreams y Receivers

- Los **DStreams** están asociados con fuentes en streaming.
- Los **Dstreams** están asociado también con objetos Receivers.
- Un **Receiver** es un objeto que recibe el dato desde una fuente y lo almacena en memoria destinada a Spark para un procesamiento posterior.
- Por cada instancia de **Dstream** va a existir un **Receiver**.
- Si queremos tener varios streams de datos en nuestra aplicación se necesitarán crear varios input **Dstream** asociados cada uno a su propio **Receiver**.

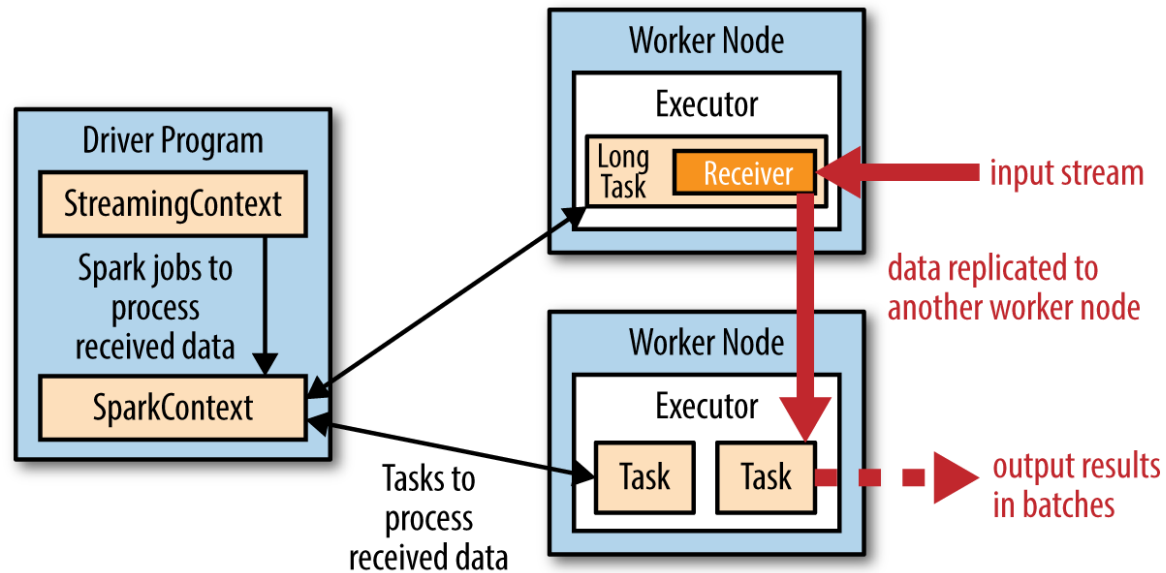
Spark Streaming – Input DStreams y Receivers

Spark Streaming
discretized stream processing



records processed in batches with short tasks
each batch is a RDD (partitioned dataset)

Spark Streaming



Spark Streaming – Recordar

- Cuando se quiere ejecutar Spark Streaming en **local** utilizar **siempre** como la master URL `local[n]`
 - Si no, si lo ejecutamos como `local` o `local[1]` existirá un único thread que se encargará de correr el receiver pero no habrá ningún thread que procese el dato
- Cuando se quiere ejecutar Spark Streaming en **cluster** se debe tener **siempre un número mayor de cores que de receivers** para poder procesar el dato proveniente de las fuentes.

Spark Streaming - Fuentes

- Dos categorías principales:
 - **Fuentes básicas:** disponibles directamente a través de la API del streaming context:
 - FileSystem
 - Socket connections
 - Akka actors
 - **Fuentes avanzadas:**
 - Kafka
 - Flume
 - Twitter
 - ...

Spark Streaming – Fuentes básicas

- **Sockets TCP:** se pueden mandar datos vía sockets a Spark Streaming.
- **Stream de ficheros:** leer ficheros desde un sistema de ficheros compatibles con la API HDFS como HDFS, S3, NFS, ...
 - Los ficheros tienen que tener el mismo formato.
 - Los ficheros pueden ser creados moviéndolos desde otro path o directamente renombrándolos.
 - Los ficheros no pueden ser modificados. Los datos nuevos que se añadan no serán procesados.
- **Stream basados en actores Akka.**
- **Stream basados en Colas de RDDs.** Útil para testing.

Spark Streaming – Fuentes básicas

- **Sockets TCP**
 - `ssc.socketTextStream(...)`
- **Stream de ficheros**
 - `streamingContext.fileStream[KeyClass, ValueClass, InputFormatClass](dataDirectory)`
 - `streamingContext.textFileStream(dataDirectory)`
- **Stream de ficherosStream basados en actores Akka.**
 - `streamingContext.actorStream(actorProps, actor-name)`
- **Stream basados en Colas de RDDs.**
 - `streamingContext.queueStream(queueOfRDDs)`

Spark Streaming – Fuentes avanzadas

- Fuentes que no están incluídas en las librerías de Spark
- Se necesitarán incluir las dependencias en el proyecto (ya visto)
- Se creará un uber JAR en las que se empaquetarán dichas dependencias dentro del proyecto que más tarde será ejecutado.
- No disponibles desde el spark-shell

Spark Streaming – Fuentes avanzadas

- Compatibilidades respecto de Spark Streaming 1.5.2:
 - Kafka 0.8.2.1
 - Flume 1.6.0
 - Kinesis Client Library 1.2.1
 - Twitter4j 3.0.3

Spark Streaming – Fuentes custom

- Implementar una clase propia Receiver.
- **Confiabilidad:**
 - **Receiver confiable:** el receiver manda una señal a la fuente confiable garantizando que no se ha perdido dato durante la ‘transacción’. Kafka y Flume permite este tipo de comunicación por ejemplo.
 - **Receiver no confiable:** no se manda ningún tipo de señal a la fuente, bien porque esta no la soporta o porque no quiere utilizar esta propiedad.
- Más info:
 - <http://spark.apache.org/docs/latest/streaming-custom-receivers.html>

Spark Streaming - Transformaciones

Transformación

Significado

Map(func)

Devuelve un nuevo Dstream donde a cada uno de los elementos se le aplica una función

flatMap(func)

Como Map, pero cada elemento de entrada puede estar mapeado a 0 ó más elementos de salida

Filter(func)

Devuelve un nuevo Dstream seleccionando únicamente los datos que cumplan el filtro indicado en la función aplicada

Repartition(numPartitions)

Permite indicar el nivel de paralelismo que se quiera utilizar a la hora de procesar el DStream

Union(otherStream)

Devuelve un nuevo Dstream unión de los elementos del Dstream fuente y los de otro stream

Spark Streaming - Transformaciones

Transformación

Significado

Count()

Devuelve un nuevo Dstream de un único elemento donde se cuentan los elementos contenidos en las RDDs

Reduce(func)

Devuelve un nuevo Dstream donde los valores de cada RDDs son agregados en relación a la función definida. Dicha función cogerá dos valores y devolverá uno.

CountByValue()

Retorna un nuevo Dstream formado por tuplas constituidas por una clave y la frecuencia de aparición de dicha clave.

reduceByKey(fun,[numTasks])

Devuelve un Dstream de clave más el agregado del valor a partir de otro Dstream clave-valor

Spark Streaming - Transformaciones

Transformación

Significado

Join(otherStream,[numTasks])

Devuelve un nuevo Dstream (K, (V,W)) a partir de dos Dstream (K,V) y (K,W)

cogroup(otherStream,
[numTasks])

Devuelve un Dstream (K, Seq[V], Seq[W]) a partir de dos Dstream (K,V) y (K,W)

transform(func)

Devuelve una nuevo Dstream que aplica una función 'de transformación' a cada RDD del Dstream origen.

updateStateByKey(func)

Devuelve un nuevo Dstream donde el estado para cada clave de los elementos del Dstream origen ha sido actualizado

Spark Streaming - Transformaciones

- A partir de una RDD de entrada [1,2,3,3]

Transformación	Ejemplo	Resultado
Map()	<code>inputRDD.map(x=>x+1)</code>	{2,3,4,4}
FlatMap()	<code>inputRDD.flatMap(x=>x.to(3))</code>	{1,2,3,2,3,3,3}
Filter()	<code>inputRDD.filter(x=> x!=1)</code>	{2,3,3}
Union()	<code>inputRDD.union({4,5})</code>	{1,2,3,3,4,5}
Count()	<code>inputRDD.count()</code>	4
countByValue()	<code>inputRDD.countByValue()</code>	{(1,1), (2,1), (3,2)}

Spark Streaming - Transformaciones

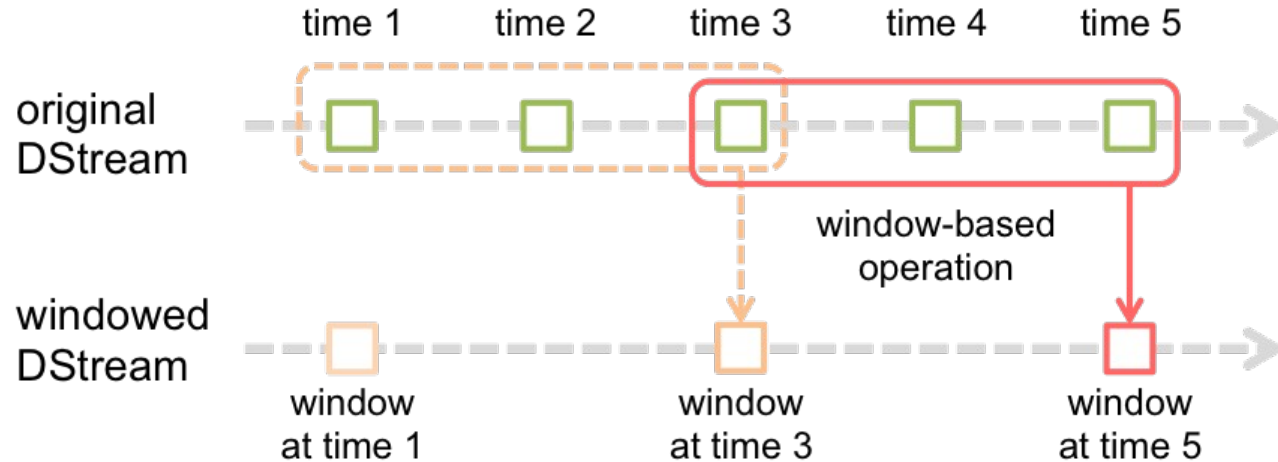
- A partir de dos pares RDD A = {(1,2), (3,4),(3,6)} y B = {(3,9)}

Transformación	Ejemplo	Resultado
reduceByKey	a.reduceByKey((x,y) => x+y)	{(1,2), (3,10)}
join	a.Join(b)	{(3,(4,9)), (3,(6,9))}
cogroup	a.Cogroup(b)	{(1,(2,[])), (3, ([4,6],9))}

Spark Streaming - Window

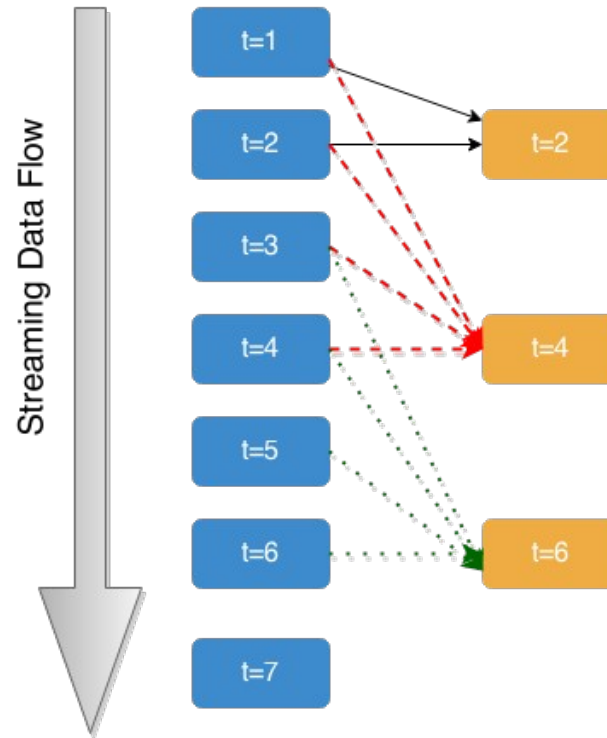
- Operaciones que permite trabajar en 'ventanas de tiempo' deslizantes
- Nos permite conocer el estado del stream de datos cada cierto tiempo.
- Las RDDs son combinadas para formar parte una RDDs perteneciente al Dstream de la ventana.
- Las operaciones window en Spark están parametrizadas por tres valores:
 - Window length
 - Sliding interval
 - Batch interval

Spark Streaming - Windowing



Spark Streaming - Windowing

Batch Interval: 1
Window duration: 4
Slide duration: 2



Spark Streaming – Windows Transformation

Transformación	Significado
window (<i>windowLength, slideInterval</i>)	Devuelve un nuevo Dstream con el resultado de computar los RDDs
countByWindow (<i>windowLength, slideInterval</i>)	Devuelve el número de elementos presentes en el intervalo de ventana
reduceByWindow (<i>func, windowLength, slideInterval</i>)	Devuelve un stream cuyo valor será el agregado definido por una función sobre los elementos en la ventana deslizando
reduceByKeyAndWindow (<i>func, windowLength, slideInterval, [numTasks]</i>)	Similar al anterior pero devuelve un Dstream de clave-valor
reduceByKeyAndWindow (<i>func, invFunc, windowLength, slideInterval, [numTasks]</i>)	El valor agregado se va calculando de manera incremental utilizando los valores 'reducidos' anteriores mediante una invFunc
countByValueAndWindow (<i>windowLength, slideInterval, [numTasks]</i>)	Devuelve un nuevo Dstream (K, Long) a partir de una entrada Dstream clave-valor

Spark Streaming – Windows Transformation

ReduceByKeyAndWindow:

```
val windowedWordCounts =  
pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a + b),  
    Seconds(30), Seconds(10))
```

Ver libro Learning Spark

UpdateStateByKey:

```
def updateRunningSum(values: Seq[Long], state: Option[Long]) =  
{ Some(state.getOrElse(0L) + values.size) }
```

```
val responseCodeDStream = accessLogsDStream.map(log =>  
    (log.getResponseCode(), 1L))  
val responseCodeCountDStream =  
responseCodeDStream.updateStateByKey(updateRunningSum _)
```

Spark Streaming – Operaciones salida

- Similar a las acciones en Spark Core
- Sirven para guardar los resultado del procesamiento en Streaming en ficheros, bases de datos, motores de búsqueda, ...
- Son las encargadas de ejecutar las transformaciones del resto del programa \Rightarrow Lazy execution
- Las operaciones de salida se ejecutan de una en una y en el orden que han sido definidas

Spark Streaming – Operaciones salida

Operación de salida	Significado
print()	Imprime los primeros diez elementos de cada batch de datos del Dstream en el nodo driver
saveAsTextFiles (<i>prefix</i> , [<i>suffix</i>])	Guarda el contenido del Dstream en un fichero de texto. La nomenclatura es: prefix-TIME_IN_MS[.suffix]
saveAsObjectFiles (<i>prefix</i> , [<i>suffix</i>])	Guarda el contenido del Dstream como un SequenceFile de objeto Java serializados. . La nomenclatura es: prefix-TIME_IN_MS[.suffix]
saveAsHadoopFiles (<i>prefix</i> , [<i>suffix</i>])	Guarda el contenido del Dstream en un fichero hadoop. La nomenclatura es: prefix-TIME_IN_MS[.suffix]
foreachRDD (<i>func</i>)	Aplica una función (en el driver) por cada RDD que compone el Dstream.

Spark Streaming – Operaciones salida

Operación	Ejemplo	Salida
Print	<code>ds.print()</code>	1 2 3 4 5 10
<code>saveAsTextFile</code>	<code>ds.saveAsTextFile(directorio)</code>	Escribirá los datos en varios ficheros en el directorio de salida
<code>foreachRDD</code>	<code>ds.foreach(rdd => rdd.saveAsHadoopFile(dir))</code>	Salva los contenidos de cada RDD a un directorio de salida

Spark Streaming – Tips foreachRDD

- Nos permite almacenar datos que previamente han sido transformados
- Se trata de una primitiva muy potente
- Existe ciertos patrones de uso de la función foreachRDD que merecen ser conocidos para optimizar la ejecución de las aplicaciones en Spark Streaming.

Spark Streaming – Patrón foreachRDD

- Caso de uso: se quiere almacenar los datos en un sistema externo que requiere crear un objeto de conexión.
- 1ª aproximación:

```
dstream.foreachRDD { rdd =>
  val connection = createNewConnection() // executed at the driver
  rdd.foreach { record =>
    connection.send(record) // executed at the worker
  }
}
```

Spark Streaming – Patrón foreachRDD

- 2ª aproximación:

```
dstream.foreachRDD { rdd =>
  rdd.foreach { record =>
    val connection = createNewConnection()
    connection.send(record)
    connection.close()
  }
}
```

- 3ª aproximación:

```
dstream.foreachRDD { rdd =>
  rdd.foreachPartition { partitionOfRecords =>
    val connection = createNewConnection()
    partitionOfRecords.foreach(record => connection.send(record))
    connection.close()
  }
}
```

Spark Streaming – Patrón foreachRDD

- Aproximación final

```
dstream.foreachRDD { rdd =>
  rdd.foreachPartition { partitionOfRecords =>
    // ConnectionPool is a static, lazily initialized pool of connections
    val connection = ConnectionPool.getConnection()
    partitionOfRecords.foreach(record => connection.send(record))
    ConnectionPool.returnConnection(connection) // return to the pool for future reuse
  }
}
```


Cacheo y persistencia

- Mismo funcionamiento que Spark Core
- Se puede cachear en memoria utilizando el método `persist()`
- Cacheo implícito en operaciones de ventana y con estado: **`reduceByWindow`**, **`reduceByKeyAndWindow`** y **`updateStateByKey`**.
- Para datos recibidos por red, el nivel de persistencia está seteado a 2 por defecto
- A diferencia de las RDDs, los Dstream se mantienen serializados en memoria por defecto.

Checkpointing

- **Caso de uso:** se tiene una aplicación Spark Streaming que va a correr por un tiempo indefinido, potencialmente de manera ilimitada.
- **Problema:** ¿qué pasa si la aplicación se cae por cualquier motivo?
- **Preguntas:** ¿vamos a perder los datos que estamos procesando?, ¿se va a poder recuperar la aplicación y trabajar sin problema?. En definitiva, ¿cuál es la tolerancia a fallos de nuestro sistema?

Checkpointing

- Igual que en el caso de Flume (canales persistentes en disco) en Spark Streaming se necesita almacenar la información necesaria para construir un sistema tolerante a fallos:
 - **Checkpointing de metadata**: se utiliza para recuperarse de fallos en el driver.
 - **Checkpointing de dato**: necesario para el funcionamiento básico al persistir RDDs que va a ser necesario en posteriores transformaciones y operaciones de salida/acciones.

Checkpointing

- **Metadatos:** almacena la información de como se va a realizar la computación en streaming en un sistema **tolerante a fallos** como HDFS. Dicha información consiste de:
 - **Configuración:** necesaria para crear la aplicación en streaming.
 - **Operaciones en el Dstream:** operaciones que definen el flujo de nuestra aplicación Spark Streaming.
 - **Batches incompletos:** trabajos incompletos que están esperando en la cola a ser ejecutados.

Checkpointing

- **Datos:** persiste las RDDs generadas.
- **Problema:**
 - RDDs intermedias con estado se necesitan cruzar en distintas partes de nuestra aplicación.
 - Spark en su 'lineage graph' lleva cuenta de todas las RDDs existentes.
 - En aplicación/computación complejas haría que dicho 'grafo' fuera poco manejable en tiempo como en tamaño.
- **Solución:**
 - Spark hace checkpointing de estas RDDs intermedias cada cierto tiempo.

Checkpointing - Configuración

```
// Function to create and setup a new StreamingContext
def functionToCreateContext(): StreamingContext = {
    val ssc = new StreamingContext(...) // new context
    val lines = ssc.socketTextStream(...) // create DStreams
    ...
    ssc.checkpoint(checkpointDirectory) // set checkpoint directory
    ssc
}

// Get StreamingContext from checkpoint data or create a new one
val context = StreamingContext.getOrCreate(checkpointDirectory, functionToCreateContext _)

// Do additional setup on context that needs to be done,
// irrespective of whether it is being started or restarted
context. ...

// Start the context
context.start()
context.awaitTermination()
```

Despliegue

- Copiar la aplicación en un cluster.
- Empaquetar la aplicación en un jar. Si se tiene dependencias externas es necesario
- Dar suficiente memoria a los executors.
- Configurar el checkpointing
- Configurar reinicio automático del driver de la aplicación.
- Configurar los write-ahead logs.
- Establecer el máximo ratio de recepción.





Tunning

- Definir el nivel de paralelismo a la hora de recibir dato.
- Definir el nivel de paralelismo a la hora de procesar dato
- Serialización de dato (Kryo vs Java serialization)
- Overhead al lanzar nuevas tareas
- Establecer el intervalo correcto de tratamiento de los batches
- Optimización del uso memoria

Otras opciones a Spark Streaming

- **Storm:** sistema de computación en tiempo real que permite procesar streams de datos en formato de tuplas.
- **Trident:** abstracción de Storm que permite computar datos con semántica exactamente una.
- **Google Dataflow:** servicio en Cloud de Google que permite procesar datos de manera batch y streaming.
- **Apache Flink:** motor de procesamiento distribuido en streaming basado en flujo de datos

Comparación de motores

Engine comparison				
				
API	MapReduce on k/v pairs	k/v pair Readers/Writers	Transformations on k/v pair collections	Iterative transformations on collections
Paradigm	MapReduce	DAG	RDD	Cyclic dataflows
Optimization	none	none	Optimization of SQL queries	Optimization in all APIs
Execution	Batch sorting	Batch sorting and partitioning	Batch with memory pinning	Stream with out-of-core algorithms

Comparación de modelos

Programing Model

	Core Storm	Storm Trident	Spark Streaming
Stream Primitive	Tuple	Tuple, Tuple Batch, Partition	DStream
Stream Source	Spouts	Spouts, Trident Spouts	HDFS, Network
Computation/Transformation	Bolts	Filters, Functions, Aggregations, Joins	Transformation, Window Operations
Stateful Operations	No (roll your own)	Yes	Yes
Output/Persistence	Bolts	State, MapState	foreachRDD

Comparación de modelos

Reliability Models

	Core Storm	Storm Trident	Spark Streaming
At Most Once	Yes	Yes	No
At Least Once	Yes	Yes	No*
Exactly Once	No	Yes	Yes*

*In some node failure scenarios, Spark Streaming falls back to at-least-once processing or data loss.

Thank you

LinkedIn
<https://es.linkedin.com/in/chicochica10>