

Práctica Procesamiento de Streams

Alumno: Araceli Macía Barrado



INDICE

1.	ESPECIFICACION DE LA PRÁCTICA.....	3
2.	DETALLE DEL SISTEMA DE PROCESAMIENTO	4
2.1	DETALLE Y COMPARATIVAS	7
2.1.1	ADQUISICIÓN DE DATOS.....	7
2.1.2	PROCESAMIENTO	9
2.1.3	ALMACENAMIENTO DE RESULTADOS.....	17
2.2	IMPLEMENTACIÓN DE CODIGO	18

1. ESPECIFICACION DE LA PRÁCTICA

El resultado de la practica debe comprender los siguientes puntos:

- a) Diseña un sistema de procesamiento con las 3 capas principales vista en clase:
 - Captura y almacenamiento intermedio
 - Procesamiento
 - Almacenamiento final y consulta
- b) Se tiene que justificar entre otras cosas porqué se ha elegido un tipo u otro de modelo de captura, porqué un procesamiento basado en eventos o micro-batching o cualquier otra decisión que creáis relevante.
- c) Justificación de un caso práctico sobre el sistema diseñado.
- d) Optativamente implementar el código del sistema construido (podéis implementar parte del sistema)

Se debe entregar un diagrama a alto nivel y 1 o 2 hojas con las justificaciones arriba anunciadas.

2. DETALLE DEL SISTEMA DE PROCESAMIENTO

El caso práctico de la arquitectura a detallar, sería el procesamiento de datos en tiempo real de Twitter.

La idea sería realizar análisis de los tweets, para mostrar datos que puedan ayudar a una empresa en la toma de decisiones.

Detallando el caso práctico por pasos:

- Ingesta de todos los tweets en idioma español
- Procesamiento de los datos con objetivo de:
 - o Realizar análisis de sentimiento de los textos.
 - o Estudiar influencias: que usuarios son favoritos, los más retwitteados o los más mencionados.
 - o Persistir los datos en Cassandra.

Posteriormente, una vez que los datos están en Cassandra se podría:

- Conectar Logstash, almacenar en Elasticsearch y visualizar un cuadro de mando en Kibana, donde una serie de indicadores se mostrarán en tiempo real. Al haber guardado todos los tweets, se podría buscar información sobre distintos usuarios o hashtag, mezclando "pasado" con el Streaming que está entrando.
- Hacer un proceso posterior de lectura de datos para hacer un estudio de tendencias por usuarios, categorizaciones de usuarios, etc.,
- Realizar una aplicación web, mostrando los datos en tiempo real, pero en un diseño personalizado y desarrollado en librerías como D3.

A continuación, se presenta el diagrama de muy alto nivel del sistema de procesamiento diseñado.



Con la arquitectura representada, se realiza la ingesta y procesamiento de datos con Spark, haciendo uso de todas las APIs que el ecosistema de Spark pone a nuestra disposición:

- Adquisición de datos,
 - o se va a realizar con el API twitterUtils que a su vez utiliza el Api Twitter4j que está desarrollado y pensando para la lectura de datos de Twitter en Streaming.
- Procesamiento:
 - o El procesamiento se realizará con el framework de Spark, lo que permitirá hacer uso de las APIs que dicho ecosistema proporciona:
 - Spark Streaming, conectado mediante el api de twitterUtils.
 - Spark Mlib: Es la librería que Spark proporciona para poder procesar los datos, por ejemplo, en este caso, se puede utilizar las funciones de tokenizar, lematizar, sobre los textos de los tweets para hacer un mínimo procesamiento de lenguaje.
 - Spark SQL permite tratar la información ingestada con lenguaje SQL. Es importante tener en cuenta que la información va cambiando en tiempo real, por lo que es importante tener un sistema que permita tener la información actualizada.

- Almacenamiento:

Para almacenamiento se ha seleccionado la base de datos NOSQL Cassandra, que tiene un conector con Spark, lo que la hace perfecta para su conexión con dicho framework.

Se ha seleccionado Spark además por sus características:

- Tolerancia a fallos
- Alto rendimiento
- Escalable

Sin embargo, estas características se repiten y se pueden encontrar en las demás tecnologías de procesamiento de datos en Streaming.

¿Por qué Spark Streaming?

Por el modo en que realiza el procesamiento, en micro-batches, en DStreams de datos. De modo que no realiza el procesamiento en tiempo real como tal, sino que durante el intervalo que se llama ventanas, va ingesting datos y almacenándolos como RDD en memoria y en la finalización de dicha ventana, captura todos estos datos en un DStream que contiene los RDD de datos ordenados y formando un conjunto que se puede procesar simultáneamente.

Esto hace que en Streaming se puedan ir haciendo operaciones en paralelo, mientras almacena los valores que van entrando en el t_0 , está realizando operaciones con el conjunto de datos que ha entrado en el $t-1$

Además, proporciona garantía de procesamiento de exactamente una vez, lo cual es muy importante, ya que ni queremos procesar los tweets más de una vez, ni queremos perder ninguno.

Tolerancia a fallos, almacenando los datos en Checkpoint en disco que hacen posible una recuperación ante errores de sistema.

Se ha escogido también por la facilidad de la programación y el desarrollo. Además, es importante destacar en este punto, la plataforma Databricks ¹ que proporciona gratuitamente un cluster de Spark sobre el que poder trabajar con esfuerzo 0 en configuración del entorno de trabajo.

2.1 DETALLE Y COMPARATIVAS

A continuación, se ofrece una breve comparativa con otras tecnologías para cada una de las capas.

2.1.1 ADQUISICIÓN DE DATOS

KAFKA:

Kafka como sistema distribuido de mensajería, es un recolector de datos desde la fuente, con muchas ventajas como que permite un almacenamiento interno para evitar la pérdida de datos.

En la arquitectura seleccionada se ha hecho uso de Spark Streaming, y dicha tecnología de procesamiento puede hacer uso de los siguientes recolectores de datos:

¹ <https://databricks.com/>



2

Entre los cuales se encuentra Kafka.

En este caso, se ha considerado que no tendría sentido generar el código de ingesta de datos Twitter en Kafka, cuando ya existe una librería accesible desde Spark para poder realizarlo.

Comentar que utilizando Kafka se podría implementar una mayor tolerancia a los fallos, puesto que se podría realizar la ingesta desde Kafka, y hacer que Spark recibiera los datos como un consumidor, de modo que utilizando almacenamiento intermedio se podría guardar los datos para poder obtenerlos en caso de que el cluster de Spark tuviera un error. Sin embargo, dado que Spark ya tiene tolerancia ante los fallos, en principio, no parece necesaria la implementación de más capas por donde hacer pasar la información de Streaming.

TWITTERUTILS

Este Api de TwitterUtils se basa en la librería: twitter4j, de hecho, lo que devuelve es el objeto twitter4j con la información, a la que se accede mediante el api de java de twitter 4j.

Ejemplo de Código:

```
val twitterStream = TwitterUtils.createStream(ssc,auth)
```

Este código devuelve un DStream de RDD de tweets. Para poder acceder a la información se utilizan los métodos del API de Twitter4j ³:

² <http://spark.apache.org/docs/latest/streaming-programming-guide.html>

³ <http://twitter4j.org/javadoc/twitter4j/Status.html>


```
val stwt= twitterStream.map(status=> (status.getId()))
```

que devolvería los ID's identificadores de cada uno de los tweets.

Como se aprecia, en pocas líneas de código, se puede recuperar fácilmente los datos.

2.1.2 PROCESAMIENTO

Spark O Storm

Storm es otra herramienta para procesar datos en tiempo real, distribuida, escalable, etc.

La diferencia, aparte de su arquitectura, es que recoge los datos en tiempo real, no utiliza micro-batching.

Aquí se puede ver una comparativa de ambas:

Storm	vs	Spark
<ul style="list-style-type: none">• Event-Streaming• At most once / At least once• sub-second• Java, Clojure, Scala, Python, Ruby• Use other tool for batch	<p>Processing Model</p> <p>Delivery Guarantees</p> <p>Latency</p> <p>Language</p> <p>Options</p> <p>Development</p>	<ul style="list-style-type: none">• Micro-Batching / Batch (Spark Core)• Exactly Once• Seconds• Java, Scala, Python• batching and streaming are very similar

4

Spark envía los datos exactamente una vez (dependiendo del uso que le demos a la ventana y la configuración del deslizamiento), sin embargo, en Storm no existe esta opción. En el caso de Twitter, en los que presumiblemente se quieren obtener los tweets, solo se necesitan los datos una vez.

Sin embargo, la latencia en Storm es menor que la latencia en Spark, pero por el contrario al trabajar con Spark Streaming dentro del ecosistema de

⁴ <https://www.slideshare.net/DavorinVukelic/realtime-streaming-with-apache-spark-streaming-and-apache-storm>

Spark, con los datos en memoria se puede realizar procesamiento analítico y machine learning con las otras librerías de Spark mientras se está recibiendo el dato.

Otra de las ventajas que se encuentran en Spark frente a Storm es en lo relacionado con el código, en Spark es mucho más intuitivo, fácil y rápido programar la ingesta desde Twitter, mientras que, con Storm, hay que programar código distinto para ingestar, para procesar y luego definir la tipología.

Ejemplo:

Código sencillo en Spark para ingestar datos:

```
val tweetStream = TwitterUtils.createStream(ssc, Utils.getAuth)
    .map(gson.toJson(_))

tweetStream.foreachRDD((rdd, time) => {
    val count = rdd.count()
    if (count > 0) {
        val outputRDD = rdd.repartition(partitionsEachInterval)
        outputRDD.saveAsTextFile(
            outputDirectory + "/tweets_" + time.milliseconds.toString)
        numTweetsCollected += count
        if (numTweetsCollected > numTweetsToCollect) {
            System.exit(0)
        }
    }
})
```

Se utiliza un api de Twitter: TwitterUtils, que devuelve DStreams. Se recorre cada uno de los RDD (conjunto de datos) que contiene un DStream y se hace una serie de operaciones con ellos, volcándolos en disco.

En Storm hay que programar la tipología.⁵

Spout: Origen de datos:

⁵ <https://github.com/davidkiss/storm-twitter-word-count/tree/master/src/main/java/com/kaviddiss/storm>

```

public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
    queue = new LinkedBlockingQueue<Status>(1000);
    this.collector = collector;

    StatusListener listener = new StatusListener() {
        @Override
        public void onStatus(Status status) {
            queue.offer(status);
        }

        @Override
        public void onDeleteNotice(StatusDeletionNotice sdn) {
        }

        @Override
        public void onTrackLimitationNotice(int i) {
        }

        @Override
        public void onScrubGeo(long l, long l1) {
        }

        @Override
        public void onStallWarning(StallWarning stallWarning) {
        }

        @Override
        public void onException(Exception e) {
        }
    };

    TwitterStreamFactory factory = new TwitterStreamFactory();
    twitterStream = factory.getInstance();
    twitterStream.addListener(listener);
    twitterStream.sample();
}

```

Se genera una clase con Spout, esta es la clase que se conecta con Twitter para la captura de datos.

Se genera un Bolt: Destino para contar las palabras:

```

public class WordCounterBolt extends BaseRichBolt {

    private static final long serialVersionUID = 2706047697068872387L;

    private static final Logger logger = LoggerFactory.getLogger(WordCounterBolt.class);

    /** Number of seconds before the top list will be logged to stdout. */
    private final long logIntervalSec;

    /** Number of seconds before the top list will be cleared. */
    private final long clearIntervalSec;

    /** Number of top words to store in stats. */
    private final int topListSize;

    private Map<String, Long> counter;
    private long lastLogTime;
    private long lastClearTime;

    public WordCounterBolt(long logIntervalSec, long clearIntervalSec, int topListSize) {
        this.logIntervalSec = logIntervalSec;
        this.clearIntervalSec = clearIntervalSec;
        this.topListSize = topListSize;
    }

    @Override
    public void prepare(Map map, TopologyContext topologyContext, OutputCollector collector) {
        counter = new HashMap<String, Long>();
        lastLogTime = System.currentTimeMillis();
        lastClearTime = System.currentTimeMillis();
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {
    }
}

```

No he puesto el código entero, pero se ve las líneas de código necesarias para procesar ahora el dato

Se genera la tipología:

```

public class Topology {

    static final String TOPOLOGY_NAME = "storm-twitter-word-count";

    public static void main(String[] args) {
        Config config = new Config();
        config.setMessageTimeoutSecs(120);

        TopologyBuilder b = new TopologyBuilder();
        b.setSpout("TwitterSampleSpout", new TwitterSampleSpout());
        b.setBolt("WordSplitterBolt", new WordSplitterBolt(5)).shuffleGrouping("TwitterSampleSpout");
        b.setBolt("IgnoreWordsBolt", new IgnoreWordsBolt()).shuffleGrouping("WordSplitterBolt");
        b.setBolt("WordCounterBolt", new WordCounterBolt(10, 5 * 60, 50)).shuffleGrouping("IgnoreWordsBolt");

        final LocalCluster cluster = new LocalCluster();
        cluster.submitTopology(TOPOLOGY_NAME, config, b.createTopology());

        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                cluster.killTopology(TOPOLOGY_NAME);
                cluster.shutdown();
            }
        });
    }
}

```

Este sería el programa principal que describe el flujo por el que pasan los datos, iniciándose en el Spout hasta llegar a procesarse en cada uno de los Bolts.

Como se ve, programar con Storm se hace más complicado en cuanto al desarrollo y mantenimiento de más líneas de código.

Storm no ofrece las funcionalidades de Map, GroupBy, join, Window.. que ofrece Spark, para utilizarlas hay que implementarlas, de ahí también la diferencia en la implementación y dificultad en el desarrollo de Storm en relación a Spark.

Es cierto que Storm tiene menor latencia que Spark, pero en relación al objetivo y necesidades del procesamiento de Tweets, la diferencia entre sub-segundos y segundos no va a tener impacto en el resultado final.

Sin embargo, teniendo en cuenta la facilidad de código, unido a que utilizar Spark da acceso directo a las otras librerías como MLlib, da más razones para escoger Spark para el procesamiento.

Flink es un nuevo framework que cada día va ganando más adeptos.

Historia de Flink

- 05/2011: Stratosphere 0.1
- 05/2014: Stratosphere 0.5 (*Apache Incubator*)
- 08/2014: Apache Flink 0.6 *Incubating*
- 01/2015: Apache Flink 0.8 *Incubating*
- 06/2015: Apache Flink 0.9
- 08/2016: Apache Flink 1.1
- 12/10/2016: Apache Flink 1.1.3

Flink al igual que Spark es un framework que permite procesamiento de Streaming y Batch y que proporcionan garantía de procesamiento de exactamente una vez.

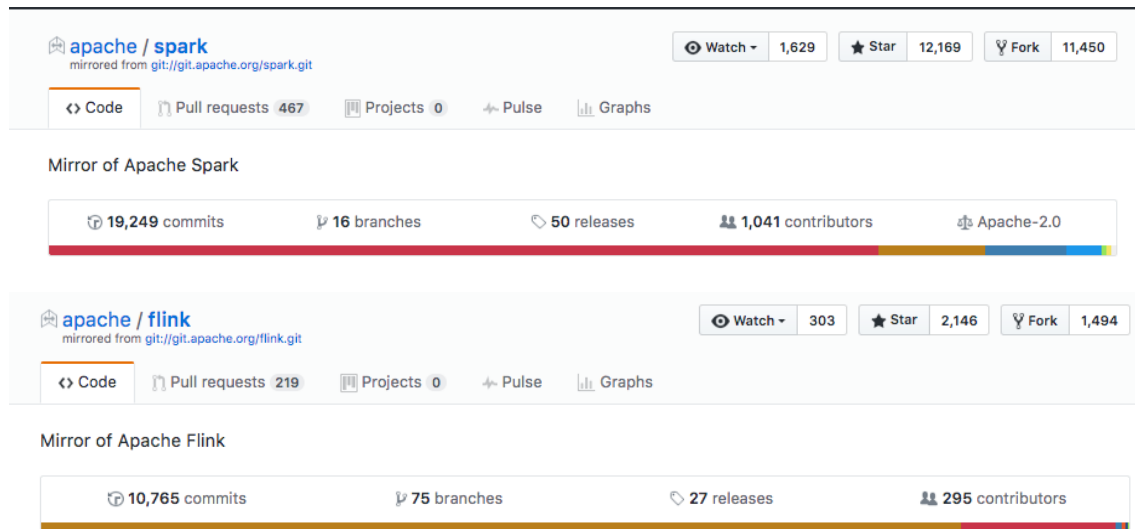
La diferencia fundamental de Flink con Spark, es que procesa los datos en tiempo real, es verdadero Streaming, mientras que ya se ha visto que Spark lo hace en micro-batches. Sin embargo, es muy difícil que el trabajar con micro-batches produzca un impacto significativo en la ingesta de los datos, a no ser que se trabaje con sistemas en tiempo real (datos de sensores o sistemas financieros) donde la falta de información al milisegundo produzca un efecto catastrófico). En el caso en que estamos que se trata de ingesta de datos desde Twitter no se va a producir un impacto por no tener el dato en el milisegundo en que se está produciendo.

Existen comparativas donde se indica que Flink es más rápido y optimiza mucho mejor los recursos que Spark, por lo que desde el punto de vista de mejorar rendimiento y latencia parece que Flink es una opción mejor.⁶

⁶ <https://www.data-blogger.com/2016/08/13/apache-flink-the-next-distributed-data-processing-revolution/>

El tema está en que en tecnologías Open Source, lo fundamental es la experiencia y que siga habiendo usuarios mejorándola y adaptándola, y en eso Spark “gana” a Flink.

Por ejemplo, en GitHub:



Existe una diferencia bastante importante entre el número de contribuyentes de una comunidad con respecto a la otra, aunque la comunidad de Flink está creciendo rápidamente, aun no llega a estar al nivel de Spark.

Experiencia de usuarios

Como se ha comentado anteriormente, una vez que se tiene claro cuál es el objetivo o la finalidad a conseguir, decidir cuál es la tecnología a utilizar depende de los factores antes mencionados:

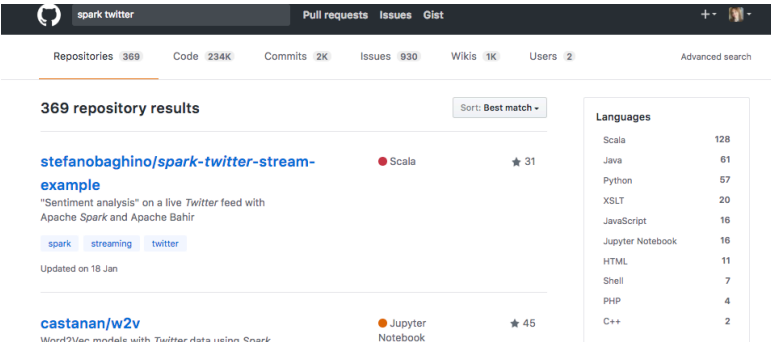
- verificar que nos ofrece garantías en cuanto a tolerancia a fallos, escalabilidad, alto rendimiento y baja latencia.

- enfrentar las casuísticas que ofrecen con los requisitos que se quieren alcanzar

Sin embargo, una vez que se hace esto y se llega a la conclusión de que con varias tecnologías podemos cumplir los requisitos técnicos y funcionales que se necesitan, lo siguiente es evaluar costes de tiempo en montar entornos de trabajo, infraestructura, curva de aprendizaje en el lenguaje de programación a utilizar, y sobre todo experiencia previa de otros desarrolladores, para no tener que inventar la rueda otra vez.

A este respecto, buscando en GitHub repositorios de desarrolladores para ver que código se podría reutilizar para la ingesta y análisis de datos en twitter, se encuentra lo siguiente:

- Repositorios de Spark, existen muchos repositorios en bastantes lenguajes de programación:



The screenshot shows a GitHub search for 'spark twitter'. The top navigation bar includes 'Pull requests', 'Issues', and 'Gist'. Below the search bar, statistics are shown: 369 Repositories, 234K Code, 2K Commits, 930 Issues, 1K Wikis, and 2 Users. The search results are sorted by 'Best match' and show 369 repository results. Two repositories are visible: 'stefanobaghino/spark-twitter-stream-example' (Scala, 31 stars) and 'castanan/w2v' (Jupyter Notebook, 45 stars). A 'Languages' sidebar on the right lists the following counts: Scala (128), Java (61), Python (57), XSLT (20), JavaScript (16), Jupyter Notebook (16), HTML (11), Shell (7), PHP (4), and C++ (2).

- Repositorios de Flink. Como ya se ha comentado antes, Flink es una framework muy bueno, pero aun esta por avanzar o expandirse en cuanto a experiencia de usuarios:

GitHub search results for 'flink twitter'. The top repository is **brakmic/TwitterFlink**, a simple Twitter-Streaming Application for Apache Flink, written in Scala, with 17 stars. It includes tags for `apache-flink`, `bigdata`, `jvm`, and `scala`. A sidebar shows the language distribution: Java (4), Scala (4), Python (1), and Shell (1).

- Repositorios de Storm, aquí se puede apreciar que es otra de las tecnologías como Spark que es ampliamente utilizada

GitHub search results for 'storm twitter'. The top repository is **P7h/StormTweetsSentimentAnalysis**, which computes sentiment analysis of tweets of US States in real-time using Storm, written in Java, with 41 stars. It includes tags for `hadoop`, `java`, `storm`, and `twitter-sentiment-analysis`. A sidebar shows the language distribution: Java (166), JavaScript (18), Python (8), Ruby (6), PHP (4), CSS (3), Jupyter Notebook (2), Scala (2), C# (1), and CoffeeScript (1).

- Repositorios de Kafka, aquí se puede apreciar que es otra de las tecnologías como Spark que es ampliamente utilizada

GitHub search results for 'kafka twitter'. The top repository is **Eneco/kafka-connect-twitter**, a Kafka Connect Sink/Source for Twitter, written in Scala, with 26 stars. It includes tags for `avro`, `connector`, `kafka`, `kafka-connect`, `stream`, and `tweets`. A sidebar shows the language distribution: Java (62), Scala (30), Python (22), JavaScript (10), HTML (4), Shell (4), Jupyter Notebook (2), Clojure (1), Ruby (1), and XSLT (1).

Según estos resultados, resulta mucho más fácil y rápido desarrollar en Spark, puesto que existen a nuestra disposición muchos más ejemplos que poder reutilizar, con la consiguiente facilidad y rapidez en el desarrollo de una solución.

2.1.3 ALMACENAMIENTO DE RESULTADOS

Para el almacenamiento y posterior consulta para análisis o visualización se ha escogido Cassandra por las siguientes razones:

- Es una base de datos No SQL sin punto único de fallo, lo que garantiza la disponibilidad.
- Tiene un conector muy potente con Spark, lo que hace perfecta su integración.
- Cassandra está optimizada para consultas y escrituras, y es lo que se necesita en esta arquitectura, un alto throughput en la escritura de los datos que se van a almacenar en Streaming.
- Unida a que Spark SQL facilita muchísimo el acceso, almacenamiento y consultas sobre las tablas de Cassandra.

2.2 IMPLEMENTACIÓN DE CODIGO

A continuación, se muestra unos fragmentos del código, muy grosso modo de lo que habría que implementar para realizar el sistema.

El código he implementado en Scala, en el framework de Spark Streaming. Para ello, se ha utilizado el entorno de trabajo que ofrece Databrick.

1) Librerías más importantes utilizadas.

```
import org.apache.spark._
import org.apache.spark.storage._
import org.apache.spark.streaming._
//Librerías para acceso a Twitter
import org.apache.spark.streaming.twitter.TwitterUtils
import twitter4j.auth.OAuthAuthorization
import twitter4j.conf.ConfigurationBuilder
//Librerías para conector de Cassandra
import com.datastax.spark.connector.cql.CassandraConnector
import com.datastax.spark.connector._
import com.datastax.spark.connector.streaming
```

2) Ejecución del Stream

```
@transient val ssc = StreamingContext.getActiveOrCreate(creatingFunc)

ssc.start() //=> Inicio del proceso.

StreamingContext.getActive.foreach { _.stop(stopSparkContext = false) } //=> Parar el
proceso de Streaming.
```

3) Implementación de la función creatingFunc

Dentro de esta función, es la que se define la ventana de ejecución del Stream.

```
val ssc = new StreamingContext(sc, slideInterval)
```

Con esta instrucción estamos indicando el intervalo de tiempo en que va ir ingestando los datos de Twitter y formando los RDD's.



El conjunto de los RDD's que se van generando en lo definido en slideInterval conforman el DStream.

Para la ejecución, slideInterval se ha definido de 5 segundos.

A continuación, se configura y se inserta la autenticación de Twitter.

```
val auth = Some(new OAuthAuthorization(new ConfigurationBuilder().build()))
```

Previamente, en código se han cargado las variables en la sesión:

```
System.setProperty("twitter4j.oauth.consumerKey", apiKey)

System.setProperty("twitter4j.oauth.consumerSecret", apiSecret)

System.setProperty("twitter4j.oauth.accessToken", accessToken)

System.setProperty("twitter4j.oauth.accessTokenSecret", accessTokenSecret)
```

Una vez que se configura la autenticación, se accede al api de Twitter.

```
val twitterStream = TwitterUtils.createStream(ssc, auth)
```

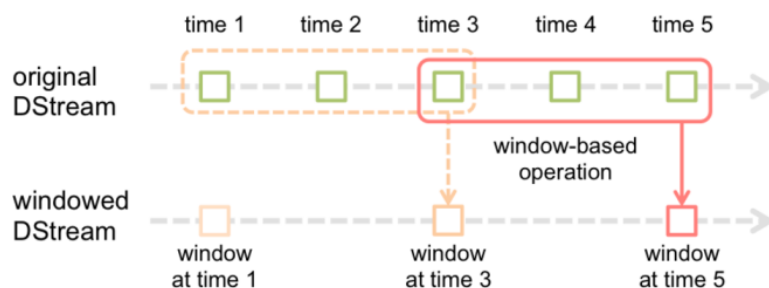
Y luego, se configura la ventana para el procesamiento:

```
val twt = twitterStream.window(windowLength, slideintervalwindowLength)
```

Aquí se indica en la variable `windowLength` el tiempo de RDD's que va unir en el procesamiento.

Para le ejecución se ha definido `windowLength` de 10 segundos.

Es decir que la ventana de ejecución va a tener los RDD's que están dentro de los 10 segundos, como `SlideInterval` era de 5 segundos, quiere decir que el `DStream` tendrá 2 Rdd's.



Otro parámetro importante es `slideintervalwindowLength`, es el tiempo en que se lanza una ventana de ejecución. Jugando con esta variable se pueden hacer ventanas de procesamiento deslizante, lo que quiere decir que hay solapamientos y Rdd's se pueden tratar en dos DStreams.

En el caso que nos aplica, el parámetro `slideintervalwindowLength` se ha configurado con el mismo valor de `windowLength`, para que no exista solapamiento.

Una vez que ya se tiene el DStream de datos, se procesan los datos de acuerdo a la funcionalidad deseada.

En este caso, lo que se hace es filtrar para dejar solo los tweets en español.

```
val twtCastellan= twt.filter(_.getLang == "es")
```

Después se hace un map, para recoger los datos que se necesiten.

Extracto del Código del map:

```
val stwt= twtCastellan.map( status=> {  
    val id=status.getId()  
    val idStr=status.getId().toString()  
    val fechaclaveparticion = dateFormat2.format(status.getCreatedAt()).toString() //FORMATO YYYY-MM-DD  
    val createdat = dateFormat.format(status.getCreatedAt()).toString()  
    val texto=status.getText()  
    val usuarioId=status.getUser().getId().toString()  
    val nombreUsuario=status.getUser().getName()  
}
```

Después se inserta la información en Cassandra:

```
stwt.foreachRDD { rdd =>{  
    var tablaTweets = rdd.map(TweetTotal=> (TweetTotal.fechaclaveparticion ,TweetTotal.idtw, 0,  
    TweetTotal.createdat, TweetTotal.idstr, TweetTotal.nombreusuario,  
    0, TweetTotal.texto,TweetTotal.usuarioId))  
    .saveToCassandra("kspacetweets", "tweets", SomeColumns("fechaclaveparticion", "idtw", "favcountorig" ,  
    "fechacreacion", "idstr", "nombreusuario" ,  
    "retweetsorig", "texto", "usuarioId"))  
    // val tablaliciarios = rdd.map(TweetTotal=> ("3622415597" , "Karina Fuentes DNS" , 588 ,2))  
}
```

Como se ha comentado antes, la relación entre Spark y Cassandra se realiza mediante un conector, y resulta muy fácil realizar la inserción, en una sola línea de código están persistidos los datos en Cassandra.

En relación al conector, comentar que hay que prestar especial atención a los nombres de las columnas, ya que debe coincidir exactamente el nombre y el

tipo. Si no coincide, no salta ninguna excepción, pero los datos no se insertarán en la tabla de la base de datos.

Por último, comentar, el modo de funcionamiento de la inserción en Cassandra:

- Si el dato no existe, lo crea, añadiéndolo a la tabla.
- Si el dato existiera, lo sobrescribe en la tabla.

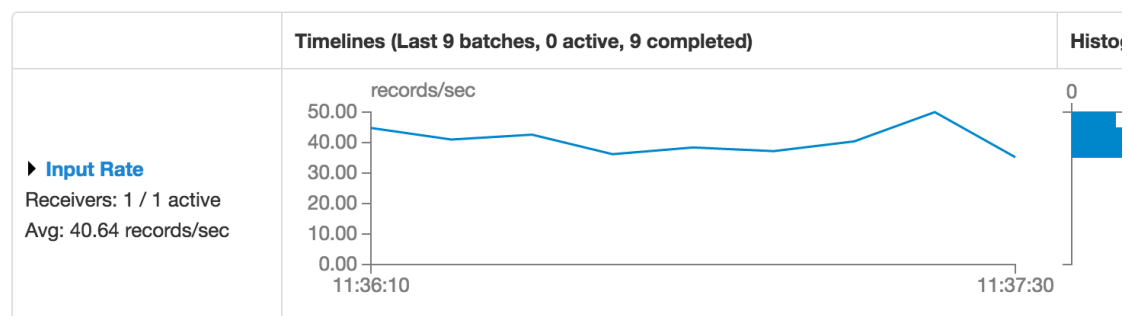
Este modo de funcionamiento se puede modificar para que no se sobrescriba la información, pero en nuestro caso, dado que la info de un tweet puede venir actualizada en un tweet si este ha sido retuiteado, resulta súper útil y ventajoso que Cassandra permita la sobreescritura de los datos de esta forma tan sencilla y fácil.

Por supuesto, este código no cubre toda la funcionalidad expuesta, pero si se intenta realizar lo principal, ingesta, tratamiento y persistencia de datos.

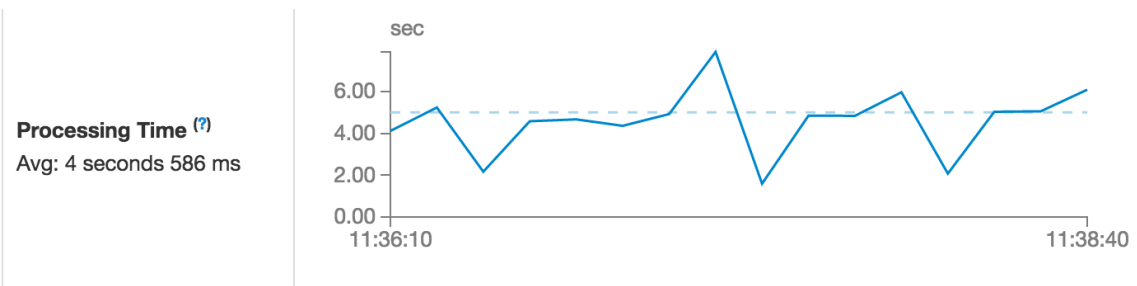
A continuación, se muestran unas graficas que se pueden obtener en el entorno de Databrick en tiempo de ejecución del Streaming.

Estadísticas de Ejecución:

Running batches of 5 seconds for 1 minute 36 seconds since 2017/03/28 11:35:55 (9 completed batches, 1829 records)



En este grafico se pueden ver que los batches son de 5 segundos (lo que se ha configurado) y que en 1 minuto con 36 segundos han llegado 1829 tweets procesados en 9 ventanas de ejecución.



El tiempo de procesamiento de cada ventana es de 4 segundos. Es importante que el tiempo en que procesa un DStream sea menor que la entrada de datos, porque si no se producirían retrasos en la cola de entrada de los datos.

También se puede ver el detalle de un proceso para determinar cuál es el proceso que esta consumiendo mas segundos, por si se puede hacer algo para optimizarlo.

Output Op Id	Description	Output Op Duration	Status	Job Id	Job Duration
0	print at <console>:152 +details	0.1 s	Succeeded	233	72 ms
				234	6 ms
				235	6 ms
1	foreachRDD at <console>:153 +details	53 ms	Succeeded	236	48 ms
2	foreachRDD at <console>:156 +details	5 s	Succeeded	237	0.6 s
				238	0.6 s
3	print at <console>:177 +details	79 ms	Succeeded	239	50 ms
				240	7 ms
				241	5 ms
4	foreachRDD at <console>:179 +details	0.5 s	Succeeded	242	0.2 s
				243	0.2 s

En este caso, las tareas que consumen más son unas líneas de escritura en el log, con lo que con esta información ya sabríamos que quitarlas sería buena idea para optimizar el tiempo de procesamiento.