Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

DISEÑO, PROGRAMACIÓN Y DESPLIEGUE DE UNA PLATAFORMA DE BIG DATA PARA EL PROCESAMIENTO DE INFORMACIÓN DE REDES SOCIALES Y WEB 2.0

TRABAJO FIN DE MÁSTER

Cayetano Rodríguez Medina

2017

Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

Máster Universitario en
Ingeniería de Redes y Servicios Telemáticos

TRABAJO FIN DE MÁSTER

DISEÑO, PROGRAMACIÓN Y DESPLIEGUE DE UNA PLATAFORMA DE BIG DATA PARA EL PROCESAMIENTO DE INFORMACIÓN DE REDES SOCIALES Y WEB 2.0

Autor
Cayetano Rodríguez Medina

Director
Borja Bordel Sánchez

Ponente
Diego Martín de Andrés

Departamento de Ingeniería de Sistemas Telemáticos

2017

Resumen

El paradigma Big Data representa el procesado de cantidades de datos tales, que resultaría imposible obtener un resultado en un tiempo razonable utilizando las técnicas de análisis convencionales. Existen diversas soluciones para Big Data, incluyendo algunas para procesar flujos en streaming, archivos de texto, o incluso colecciones de información heterogénea.

Los usuarios están permanentemente conectados a sus dispositivos digitales (ordenadores, teléfonos móviles, tablets...), por lo que el consumo de las redes sociales es cada vez mayor. El análisis de las mismas permite acceder a información realmente valiosa sobre el público objetivo del negocio, los clientes potenciales, la competencia, las tendencias de consumo o de mercado, etc.

El Big Data aplicado a las redes sociales consiste en la monitorización y medición de los datos que circulan por las redes de una empresa. A mayor diversificación en las redes sociales, mayores serán los esfuerzos implementados en la estrategia de Big Data. Dentro de la web 2.0 se mueve una inmensa cantidad de información, pero cada negocio decide donde centrar su atención según sus intereses. Lo más importante es comprender que las redes sociales forman parte del Big Data actualmente. Esto es, que las redes sociales son la principal fuente de información para el Big Data. Y no solo el contenido es información, sino también visualizaciones, "likes", "shares", seguidores, "retweets", comentarios, descargas... Las redes sociales permiten extraer una cantidad de información inimaginable.

En este trabajo se deberá implementar una plataforma de procesamiento de datos, donde se analice la información proveniente de varias redes sociales (principalmente Twitter al disponer de una interfaz para ello) y de la web 2.0 (por ejemplo, Google Scholar). Para ello se hará uso de técnicas de Big Data y se emplearán las soluciones tecnológicas de Apache Spark. Sobre una infraestructura de computación en la nube se desplegará una plataforma de procesamiento donde se combinen algunas de las tecnologías más altamente demandadas en la actualidad como Jupyter, Apache Kafka o Apache ZooKeeper.

Se deberá desarrollar una solución plenamente funcional sobre la que se implementarán algoritmos para segmentar y microsegmentar a los miembros de una red social (con vistas a banca, marketing tecnológico, etc.), para detectar relaciones y posiciones de poder, y para obtener cualquier otra información relevante que constituya el negocio de las grandes compañías tecnológicas actuales (Facebook, Google, etc.).

Abstract

The Big Data paradigm represents the processing of quantities of data such that it would be impossible to obtain any results in a reasonable time using conventional analysis techniques. There are various solutions for Big Data, including some for processing streaming streams, text files, or even collections of heterogeneous information.

Users are permanently connected to their digital devices (computers, mobile phones, tablets ...), so the consumption of social networks is increasing. The analysis of them grants access to really valuable information about the target audience of the business, potential customers, competitors, trends in consumption or market, etc.

The Big Data applied to social networks is based on the monitoring and measurement of the data that circulate through the networks of a company. The greater the diversification in social networks, the greater the efforts implemented in the Big Data strategy. Within the web 2.0 a huge amount of information moves, but each business decides where to focus their attention according to their interests. The most important thing is to understand that social networks are currently part of Big Data. That is, social networks are the main source of information for Big Data. And not only content is information, but also visualizations, likes, shares, followers, retweets, comments, downloads... Social networks allow you to extract a quantity of information unimaginable.

In this work, a data processing platform should be implemented, analyzing the information coming from various social networks (mainly Twitter, having an interface for it) and web 2.0 (for example, Google Scholar). This will use Big Data techniques, and Apache Spark technology solutions will be used. A cloud computing infrastructure will deploy a processing platform that combines some of the most highly demanded technologies today like Jupyter, Apache Kafka or Apache ZooKeeper.

A fully functional solution has to be developed on which algorithms will be implemented to segment and microsegmentate the members of a social network (with a view to banking, technological marketing, etc.), to detect relations and positions of power, and to obtain any other relevant information that constitutes the business of today's big technology companies (Facebook, Google, etc.).

Índice general

Índice de figuras

Siglas

AMI	Amazon Machine Image
API	Application Programming Interface
AWS	Amazon Web Services
CPU	Central Processor Unit
EBS	Elastic Block Store
EC2	Elastic Cloud Compute
IDE	Integrated Development Environment
JAR	Java ARchive
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
SSH	Secure SHell
VPC	Virtual Private Cloud

1 Introducción

El paradigma Big Data representa el procesado de cantidades de datos tales, que resultaría imposible obtener un resultado en un tiempo razonable utilizando las técnicas de análisis convencionales. Existen diversas soluciones para Big Data, incluyendo algunas para procesar flujos en streaming, archivos de texto, o incluso colecciones de información heterogénea.

1.1 Motivación

El concepto de Big Data aplica para toda aquella información que no puede ser procesada o analizada utilizando procesos o herramientas tradicionales. Sin embargo, Big Data no se refiere a alguna cantidad en específico, ya que es usualmente utilizado cuando se habla en términos de petabytes y exabytes de datos.

Además del gran *volumen* de información, esta existe en una gran *variedad* de datos que pueden ser representados de diversas maneras en todo el mundo, por ejemplo de dispositivos móviles, audio, video, sistemas GPS, incontables sensores digitales en equipos industriales, automóviles, medidores eléctricos, veletas, anemómetros, etc., de tal forma que las aplicaciones que analizan estos datos requieren que la *velocidad* de respuesta sea lo demasiado rápida para lograr obtener la información correcta en el momento preciso. Estas son las características principales de una oportunidad para Big Data.

Los seres humanos estamos creando y almacenando información constantemente y cada vez más en cantidades astronómicas. Esta contribución a la acumulación masiva de datos la podemos encontrar en diversas industrias, las compañías mantienen grandes cantidades de datos transaccionales, reuniendo información acerca de sus clientes, proveedores, operaciones, etc., de la misma manera sucede con el sector público.

Los usuarios están permanentemente conectados a sus dispositivos digitales (ordenadores, teléfonos móviles, tablets...), por lo que el consumo de las redes sociales es cada vez mayor. El análisis de las mismas permite acceder a información realmente valiosa sobre el público objetivo del negocio, los clientes potenciales, la competencia, las tendencias de consumo o de mercado, etc.

Por ejemplo, en el caso de Twitter, en media, cada segundo, se generan 6.000 tweets, lo cual corresponde a 350.000 tweets enviados por minuto, 500 millones de tweets al día, y alrededor de 200 billones (americanos, o lo que es lo mismo, 200 mil millones) de tweets por año. En el caso de Facebook, cada minuto se publican unos 510.000 comentarios, 293.000 estados son actualizados, se suben unas 243.000 fotos (unas 350

millones de fotos al día), y se generan 4 millones de "likes". Se estima que Facebook genera 4 nuevos petabytes de datos cada día.

Estos datos de dos de las redes sociales más usadas actualmente, nos muestran las enormes cantidades de datos generados por las redes sociales en cada segundo. Todos esos datos pueden convertirse en información muy valiosa, gracias a las técnicas de Big Data.

El Big Data aplicado a las redes sociales consiste en la monitorización y medición de los datos que circulan por las redes. A mayor diversificación en las redes sociales, mayores serán los esfuerzos implementados en la estrategia de Big Data. Dentro de la web 2.0 se mueve una inmensa cantidad de información, pero cada negocio decide dónde centrar su atención según sus intereses. Lo más importante es comprender que las redes sociales forman parte del Big Data actualmente. De hecho, las redes sociales son la principal fuente de información para el Big Data. Y no solo el contenido es información, sino también las visualizaciones, "likes", "shares", seguidores, "retweets", comentarios, descargas... Las redes sociales permiten extraer una cantidad de información inimaginable.

1.2 Objetivos

Los objetivos definidos para el presente trabajo, son los que se presentan a continuación:

- Implementación de una plataforma de procesamiento de datos
- Análisis de información proveniente de redes sociales y de la web 2.0
- Despliegue de la plataforma de procesamiento sobre una infraestructura de computación en la nube
- Implementación de algoritmos para segmentar a los miembros de una red social, así como para obtener otra información de interés

1.3 Estructura de la memoria

El presente documento describe la implementación de una plataforma de procesamiento de datos en la nube, para analizar información proveniente de redes sociales y web 2.0. Para lo cual, se ha seccionado en los siguientes capítulos:

- **Capítulo 1 – *Introducción***: se describe el contexto actual en el que se ubica el Big Data, especialmente el enfocado a las redes sociales, aportando las motivaciones y objetivos principales del trabajo.

- **Capítulo 2 – *Tecnologías utilizadas***: descripción de las principales tecnologías utilizadas a lo largo del proyecto, incluyendo descripción teórica de las mismas y funcionamiento básico.
- **Capítulo 3 – *Fuentes de datos***: introducción a las fuentes que se usarán para extraer grandes cantidades de datos que serán procesados y analizados posteriormente.
- **Capítulo 4 – *Implementación de la plataforma de procesamiento de datos***: se describe el proceso seguido para la implementación y despliegue de una plataforma de procesamiento de datos sobre una infraestructura de computación en la nube, combinando diversas tecnologías.
- **Capítulo 5 - *Introducción de mejoras en la plataforma de procesamiento de datos:*** en el apartado anterior diseñamos e implementamos una plataforma de procesamiento de datos básica. A partir de esta, se decidió introducir una serie de mejoras que consiguieran una solución más completa y escalable.
- **Capítulo 6 – *Procesamiento de datos y análisis de información***: desarrollo e implementación de aplicaciones y algoritmos para procesar los datos extraídos de las diversas fuentes, así como el posterior análisis de la información obtenida de dicho procesamiento.
- **Capítulo 7 – *Conclusiones***: finalmente en el capítulo se describen las conclusiones obtenidas del estudio del proyecto, así como la experiencia de uso de las tecnologías de Big Data.
- **Bibliografía**
- **Anexos**

Podemos distinguir dos bloques principales: el primero introductorio, incluyendo los capítulos 1 y 2, enfocado a contextualizar el trabajo y a presentar de una manera breve y concisa las tecnologías que se usarán posteriormente; y el segundo, formado por los capítulos 3, 4 y 5, y 6, cada uno de los cuales se asocia a uno de los bloques principales del Big Data (fuentes de datos, plataforma de datos y análisis de datos). Por último, el capítulo 7 muestra las conclusiones sacadas de este proyecto. En esta estructura se refleja la principal división del proyecto en dos partes principales: la implementación de la plataforma de datos, y el procesamiento y análisis de los datos.

2 Tecnologías utilizadas

Para la realización de este trabajo, ha sido necesario utilizar diversas tecnologías de gran interés, especialmente en el ámbito del Big Data. Se considera necesario realizar una introducción a estas tecnologías, previamente a la descripción del proceso de implementación de la plataforma de procesamiento de datos.

2.1 Apache Spark

Apache Spark es un sistema de computación de clúster rápido y de propósito general. Proporciona APIs de alto nivel en Java, Scala, Python y R. De hecho, es posible crear aplicaciones interactivamente a través de las shells de Scala, Python y R.

Figura 1. Logo de Apache Spark

También soporta un gran conjunto de herramientas de alto nivel, como Spark SQL para SQL y procesamiento de datos estructurados, MLlib para el aprendizaje automático, GraphX para procesamiento de gráficos y Spark Streaming. Es posible combinar todas estas librerías sin problemas en la misma aplicación.

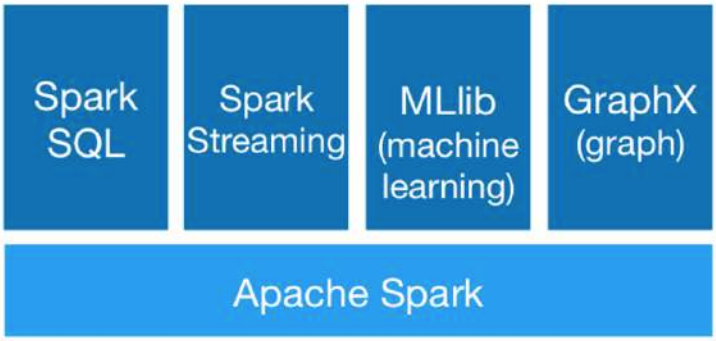

Figura 2. Módulos de Apache Spark

Podemos ejecutar Spark usando su modo "standalone cluster" (*cluster autónomo*), sobre EC2 (en la nube de Amazon Web Services), sobre Hadoop YARN, o sobre Apache Mesos. Puede acceder a diversas fuentes de datos, incluyendo HDFS, Cassandra, HBase, S3, Hive, Tachyon o cualquier otra fuente de datos en Hadoop.

2.1.1 Modo Cluster

Existen varias opciones para ejecutar Spark sobre un clúster. Es posible que Spark ejecute ambas (tanto el clúster como sí mismo), o bien, ejecutarse sobre otros gestores de clusters, como Apache Mesos o Hadoop YARN.

Las aplicaciones de Spark se ejecutan como un conjunto de procesos independientes sobre un cluster, coordinados por el objeto ***SparkContext*** localizado en nuestro programa principal (conocido como ***driver program***).

Específicamente, para ejecutar Spark sobre un cluster, el SparkContext puede conectarse a distintos gestores de clusters (el cluster autónomo propio de Spark, Apache Mesos o Hadoop YARN), que distribuyen los recursos entre las distintas aplicaciones. Una vez conectado, Spark obtiene ***executors*** en los nodos del cluster, que son procesos que ejecutan computaciones y almacenan información de las aplicaciones. A continuación, envía el código de la aplicación (definido por un JAR o ficheros Python pasados al SparkContext) a los executors. Finalmente, el SparkContext envía ***tasks*** a los executors para que las realicen.

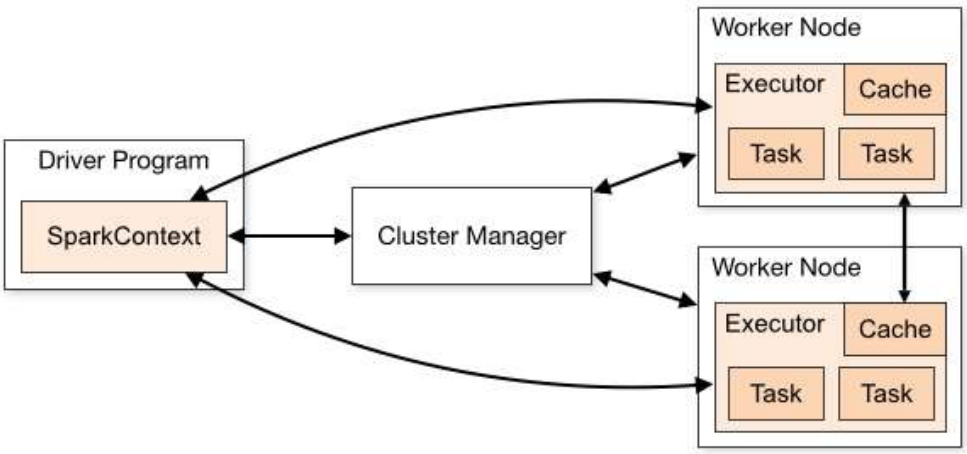

Figura 3. Spark: modo cluster

Algunos aspectos de interés con respecto a esta arquitectura:

1) Cada aplicación tiene sus propios procesos executors, que se mantendrán durante la duración de la aplicación completa, y ejecutarán tasks en múltiples hebras. Esto conlleva el beneficio de tener aplicaciones aisladas unas de otras, en el lado del planificador (cada driver planifica sus propias tasks) y en el lado del executor (tasks de distintas aplicaciones se ejecutan en diferentes JVMs). De todas formas, también conlleva que los datos no pueden ser compartidos entre diferentes aplicaciones de Spark (instancias de SparkContext) sin llevarlos a un sistema de almacenamiento externo.

2) Spark es agnóstico con respecto al gestor de cluster subyacente. Mientras pueda obtener procesos executors, y estos puedan comunicarse entre ellos, es relativamente fácil ejecutarlo incluso sobre un gestor de clusters que también soporte otras aplicaciones (como Mesos o YARN).
3) El driver program debe escuchar y aceptar conexiones entrantes de sus executors durante todo su lifetime. Por tanto, el driver program debe ser direccionable desde los nodos workers.
4) Debido a que el driver planifica tasks en el cluster, debería ejecutarse cerca de los nodos workers, preferiblemente en la misma red de área local.

Spark actualmente soporta tres gestores de cluster:

- **Standalone** (autónomo): un gestor de cluster sencillo incluido en Spark que hace fácil montar un cluster.
- **Apache Mesos**: un gestor de cluster general que también puede ejecutar Hadoop MapReduce y aplicaciones de servicios.
- **Hadoop YARN**: el gestor de recursos en Hadoop 2.

2.1.2 RDDs (*Resilient Distributed Datasets*)

Spark funciona alrededor del concepto de **RDDs** (*Resilient Distributed Datasets*), que son colecciones de elementos capaces de operar en paralelo. Presentan tolerancia a fallos. Existen dos formas de crear un RDD: **paralelizando una colección** ya existente en el programa del driver, o bien, referenciando un dataset en un **sistema de almacenamiento externo**, como un sistema de ficheros compartido, HDFS, HBase, Amazon S3, Cassandra, o cualquier otra fuente de datos soportada por Hadoop.

Las colecciones paralelizadas se crean copiando los elementos de una colección ya existente en el programa, para formar un dataset distribuido que puede ser operado en paralelo. Un parámetro muy importante en las colecciones paralelizadas es el número de particiones en el que se divide el cluster. Spark ejecutará una tarea por cada partición del cluster. Típicamente, habrá 2 - 4 particiones por cada CPU en el cluster. Lo habitual es que Spark trate de configurar el número de particiones automáticamente basándose en el cluster. De todas formas, siempre es posible especificarlo manualmente. Sin embargo, lo más normal es cargar los datos de una fuente de datos externa.

Los RDDs soportan dos tipos de operaciones: **transformaciones**, que crean un nuevo dataset a partir de otro existente, y **acciones**, que devuelven un valor tras ejecutar una computación sobre el dataset. Todas las transformaciones en Spark se computan siguiendo un paradigma muy utilizado en Big Data: ***Lazy Evaluation***. Esto significa que las transformaciones sólo se computan cuando se vaya a utilizar el RDD resultante en una acción.

Spark permite guardar RDDs para reutilizarlos varias veces. Cuando se persiste un RDD, cada nodo almacena sus particiones y las reutiliza en otras acciones del RDD (o en otros RDDs derivados de este). Esto es interesante para algoritmos iterativos y uso interactivo rápido. La primera vez que se compute para una acción, se guardará en la memoria de los nodos. La caché de Spark es tolerante a fallos, de modo que si una partición del RDD se pierde, se volverá a computar automáticamente usando las transformaciones que originalmente lo crearon. Además, cada RDD se puede almacenar usando un *nivel de almacenamiento* distinto.

En resumen, usaremos los RDDs en un programa de Spark de la siguiente forma:

1) Crear RDDs de entrada a partir de fuentes de datos externas (o a partir de una colección ya existente).
2) Transformar estos RDDs para obtener nuevos datasets.
3) Almacenar cualquier RDD intermedio que necesitemos reutilizar.
4) Lanzar acciones sobre los RDDs para empezar una computación paralela, que es optimizada y ejecutada por Spark.

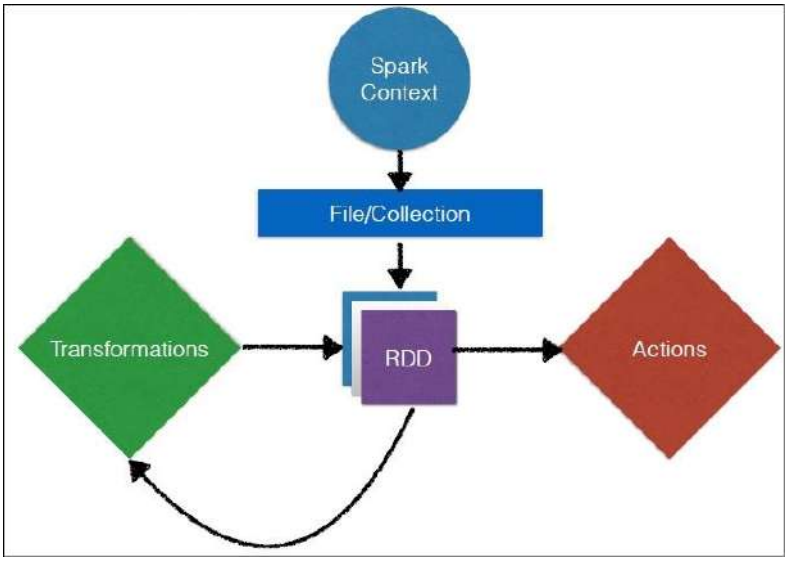

Figura 4. Ciclo de vida de un RDD

2.1.3 Spark SQL

Spark SQL es un módulo de Spark para el procesamiento de datos estructurados (o semiestructurados). Con datos estructurados se entiende datos que tienen un esquema, es decir, un conjunto consistente de campos a lo largo de todos los registros.

Spark SQL proporciona 3 funcionalidades principalmente:

1) Proporciona una abstracción de **DataFrame** para Python, Java y Scala, que simplifica el trabajo con datasets estructurados. Los DataFrames son como las tablas en una base de datos relacional.

2) Puede leer y escribir datos en una gran **variedad de formatos estructurados** (ej.- JSON, Hive y Parquet).

3) Permite hacer **consultas a los datos usando SQL**, tanto desde un programa de Spark como desde herramientas externas que se conecten a Spark SQL a través de conectores de bases de datos estándar (JDBC/ODBC).

Spark SQL se basa en una extensión del modelo RDD, llamada DataFrame. Un DataFrame contiene un RDD de objetos Row (*fila*), cada uno representando un registro. Un DataFrame también conoce el esquema de sus filas. Estos almacenan los datos de una forma más eficiente que los RDDs nativos, aprovechándose de su esquema. Además, proporcionan nuevas operaciones no disponibles en RDDs, como la habilidad de ejecutar queries SQL. Los DataFrames pueden crearse a partir de fuentes de datos externas, del resultado de consultas, o de RDDs normales.

2.1.4 Spark Streaming

Spark Streaming es una extensión del core Spark API que permite el procesamiento de streaming de flujos de datos en vivo, escalable, de alto rendimiento y tolerante a fallos. Los datos se pueden ingerir de diversas fuentes como Kafka, Flume, Kinesis, o sockets TCP, y se pueden procesar usando algoritmos complejos expresados con funciones de alto nivel como `map`, `reduce`, `join` y `window`. Finalmente, los datos procesados se pueden enviar a sistemas de ficheros, bases de datos y dashboards. De hecho, es posible aplicar algoritmos de machine learning y procesamiento de grafos de Spark sobre los flujos de datos.

Figura 5. Spark Streaming

Internamente, funciona de la siguiente forma: Spark Streaming recibe flujos de datos de entrada en vivo, y los divide en lotes, que son procesados por el Spark Engine para generar el flujo final de resultados en lotes.

Figura 6. Flujo de datos en Spark Streaming

Spark Streaming proporciona un nivel de abstracción de alto nivel llamado *discretized stream* o **DStream**, que representa un flujo continuo de datos. Los DStreams pueden ser creados a partir de flujos de datos de entrada de las distintas fuentes, o bien aplicando operaciones de alto nivel sobre otros DStreams. Internamente, los DStreams se representan como una secuencia de RDDs. Cada RDD en un DStream contiene datos para un cierto intervalo, como se puede observar en la siguiente figura.

Figura 7. DStream

El proceso habitual seguido para diseñar un programa usando Spark Streaming es el siguiente:

1) Definir las fuentes de entrada, creando DStreams de entrada.
2) Definir las computaciones sobre el streaming, aplicando transformaciones y operaciones de salida sobre los DStreams de entrada.
3) Comenzar a recibir datos y procesarlos.
4) Esperar a que el procesamiento acabe (manualmente o debido a un error).

2.2 Jupyter Notebook

Jupyter Notebook es una aplicación web que permite crear y compartir documentos, conocidos como notebooks, que contienen código, ecuaciones, visualizaciones y texto aclaratorio. Algunas de las funciones que incluye son limpieza y transformación de datos, simulación numérica, modelado estadístico, y aprendizaje automático.

Figura 8. Logo Jupyter

Los notebooks soportan gran variedad de lenguajes de programación, incluyendo los más populares en la ciencia de datos como Python, R o Scala. Es posible compartir notebooks mediante correo electrónico, Dropbox, GitHub o la herramienta "*Jupyter Notebook Viewer*". Además, los notebooks son totalmente interactivos. El código puede generar imágenes, vídeos, LaTeX o JavaScript. Esto nos permite visualizar y manipular datos en tiempo real.

Jupyter se integra muy bien con el Big Data, potenciando otras herramientas como Apache Spark, usando Python, R o Scala. También permite explorar esos datos con *pandas, scikit-learn, ggplot2, dplyr*, etc.

2.2.1 Arquitectura

Jupyter Notebook se basa en un conjunto de estándares abiertos para la computación interactiva. Una especie de HTML y CSS para la computación interactiva en la web. Estos estándares pueden ser aprovechados por los desarrolladores de terceros para crear aplicaciones personalizadas con la computación interactiva integrada.

Los notebooks de Jupyter son un formato de documento abierto basado en JSON. Contienen un registro completo de las sesiones del usuario, así como código embebido, texto, ecuaciones y resultados enriquecidos.

El notebook se comunica con los Kernels de computación usando el ***Interactive Computing Protocol*** (*Protocolo de Computación Interactiva*), un protocolo de red abierto basado en datos JSON sobre ZMQ y WebSockets. Los Kernels son procesos que

ejecutan código interactivo en un lenguaje de programación concreto, y devuelven la salida al usuario. Estos Kernels también responden a autocompletado con el tabulador y peticiones de introspección.

2.3 Anaconda

Anaconda es un gestor de paquetes y gestor de entornos, gratuito y fácil de instalar, que además incluye una distribución de Python y una gran colección de paquetes open source. Fue creado por *Continuum Analytics*, y está especialmente diseñado para el desarrollo de programación científica. La distribución incluye instaladores automáticos para los principales sistemas operativos (Windows, Linux, OS X), y soporta dos versiones distintas del lenguaje Python (Python 2.7 y Python 3.6).

Figura 9. Logo Anaconda

Como una distribución integrada de Python, Anaconda incluye varios componentes en la misma instalación, destacando:

- ***conda***: es el gestor de paquetes para la administración y configuración de todas las librerías Python en Anaconda.
- ***Jupyter***: Anaconda instala Jupyter en el sistema, de modo que ya podamos usarlo para crear notebooks.
- ***Spyder***: IDE autónomo para el desarrollo de Python. A diferencia de Jupyter, este es un entorno de desarrollo más tradicional, con un gestor de proyectos y dependencias, un depurador (*debugger*) y diferentes intérpretes de Python para ejecutar los programas.

2.4 Amazon Web Services

Amazon Web Services ofrece recursos y servicios de cloud computing bajo demanda, conforme a un modelo de precios de pago por uso. Son muchos los servicios que ofrece AWS, aunque el que más nos interesa para la realización de este proyecto es el servicio ***EC2*** (*Elastic Cloud Compute*).

Figura 10. Logo de Amazon Web Services

2.4.1 Amazon EC2

Amazon Elastic Cloud Compute (EC2) ofrece capacidades de computación escalables en la nube pública de Amazon. Este servicio nos permite montar servidores en la nube de Amazon, eliminando la necesidad de adquirir recursos hardware. Gracias a esto, podemos desarrollar y desplegar aplicaciones mucho más rápido.

Podemos lanzar tantas instancias de servidores virtuales como queramos, configurando su seguridad y redes, y gestionando el almacenamiento. También nos permite escalabilidad, lo cual nos permite manejar los cambios en los requisitos y los picos de popularidad, y reduciendo la necesidad de realizar previsiones de tráfico.

Amazon EC2 ofrece las siguientes funcionalidades:

- Entornos de computación virtuales, conocidos como ***instancias***.
- Plantillas preconfiguradas para las instancias, conocidas como ***Amazon Machine Images*** (*AMIs*), que empaquetan todo lo necesario para el servidor, incluyendo el sistema operativo y software adicional.
- Varias configuraciones de CPU, memoria, almacenamiento, y capacidades de red para las instancias, conocidas como ***tipos de instancia***.
- Información de login segura para las instancias, usando ***key pairs*** (AWS almacena la clave pública, y el usuario guarda la clave privada).
- Volúmenes de almacenamiento para datos temporales que se eliminan cuando se para o termina una instancia, conocidos como ***volúmenes de almacenamiento de instancias***.
- Volúmenes de almacenamiento persistente para las instancias, usando el servicio *Amazon Elastic Block Store* (*EBS*), conocidos como ***volúmenes Amazon EBS***.
- Múltiples localizaciones físicas para los recursos del usuario, como las instancias o los volúmenes EBS, conocidas como ***regiones*** y ***zonas de disponibilidad*** (***Availability Zones***).

- Un firewall que permite al usuario especificar los protocolos, puertos y rangos de direcciones IP que pueden alcanzar la instancia, usando los ***grupos de seguridad***.
- Direcciones IPv4 estáticas para cloud computing dinámico, conocidas como ***direcciones IP elásticas***.
- Metadatos, conocidos como ***etiquetas***, que se pueden crear y asignar a los recursos de Amazon EC2.
- Redes virtuales que se pueden crear y que están aisladas lógicamente del resto de la nube de AWS, y que se pueden conectar opcionalmente a tu red propia, conocidas como ***Virtual Private Clouds*** (*VPCs*).

2.5 Apache Kafka

Kafka es un sistema de almacenamiento distribuido basado en los sistemas publicador/suscriptor (también conocidos como sistemas basados en eventos). El fundamento de este tipo de sistemas es que un gran número de productores (publicadores) distribuyen información de interés (eventos) a un gran número de consumidores (suscriptores).

Figura 11. Logo Apache Kafka

Es una especie de servicio de mensajería pensado como un registro distribuido de logs. En primer lugar, vamos a comentar un poco sobre la terminología de mensajería en Kafka:

- Kafka almacena los mensajes por categorías llamadas ***topics***.
- A los procesos que publican mensajes en un topic de Kafka, se les denomina ***productores*** (*producers*).
- A los procesos que se suscriben a los topics y leen los mensajes publicados, se les denomina ***consumidores*** (*consumers*).
- Kafka se ejecuta como un clúster compuesto por uno o más servidores, cada uno de los cuales se denomina ***broker***.

Básicamente, los productores envían mensajes a través de la red al clúster de Kafka, el cual los sirve a los consumidores de la siguiente forma:

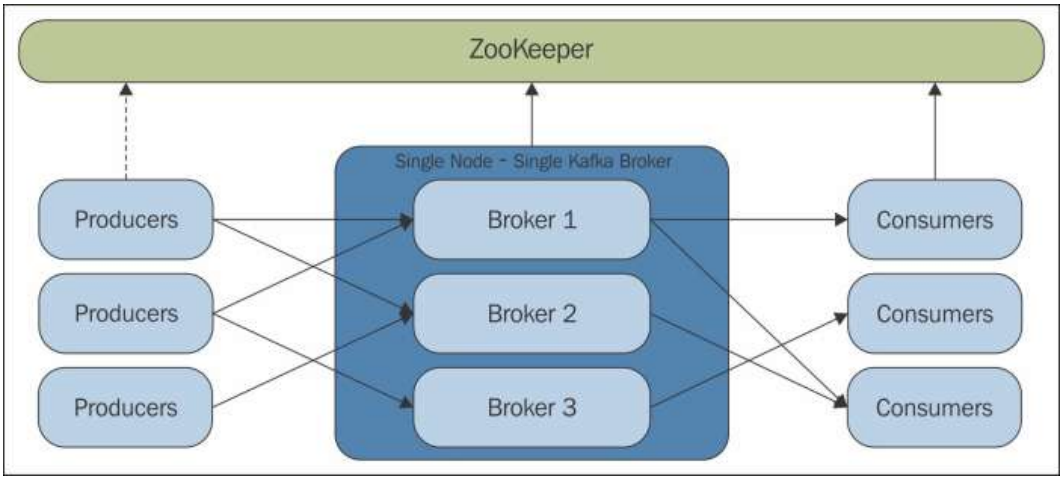

Figura 12. Esquema funcionamiento general de Kafka

La comunicación entre clientes y servidores, es llevada a cabo mediante un protocolo TCP independiente del lenguaje de programación. Apache proporciona un cliente Java para Kafka, pero hay clientes en muchos lenguajes distintos.

2.5.1 Apache ZooKeeper

Apache ZooKeeper es un servicio de coordinación distribuida muy rápido, con alta disponibilidad y tolerancia a fallos. Se usa en Kafka para la gestión y coordinación de los broker de Kafka. Cada broker se comunica y coordina con otros broker del clúster usando ZooKeeper. Los productores y consumidores son avisados por el servicio ZooKeeper cuando aparece un nuevo broker en el sistema o cuando se cae alguno de ellos.

3 Fuente de datos: Twitter

El elemento de entrada de cualquier plataforma de Big Data es el formado por las distintas fuentes de datos. Los datos que serán procesados tendrán que ser extraídos en primer lugar de algún sitio, para pasar posteriormente por la plataforma de procesamiento. En nuestro caso, la principal fuente de datos será la red social Twitter, debido a que posee una API que puede ser utilizada abiertamente y de forma gratuita por cualquier usuario.

Twitter es una de las redes sociales más conocidas y usadas actualmente. El servicio que ofrece Twitter se conoce como *microblogging*, que es una variante del *blogging* en el que los trozos de contenido son extremadamente pequeños. En el caso de Twitter, hay una limitación de 140 caracteres para cada tweet. En contraposición a otras redes sociales como Facebook, la red de Twitter no es bidireccional, lo cual quiere decir que las conexiones no tienen por qué ser mutuas: tú puedes seguir a otros usuarios que no te sigan y viceversa. Dada la variedad de usos que tiene, Twitter es una potencial mina de oro para el Big Data.

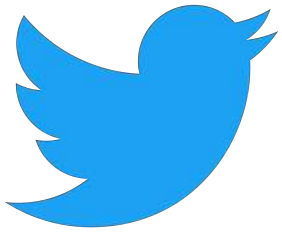

Figura 13. Logo Twitter

Twitter ofrece una serie de APIs para proveer acceso programático a los datos de Twitter, incluyendo leer tweets, acceder a perfiles de usuarios, o publicar contenido en nombre de un usuario.

Con el fin de configurar nuestro proyecto para que acceda a los datos de Twitter, existen dos pasos preliminares:

- Registrar nuestra aplicación
- Seleccionar una API

3.1 Registro de aplicación y Autenticación

Registrar la aplicación es muy sencillo. Tan sólo hay que acceder a la siguiente URL: https://apps.twitter.com/, habiendo iniciado sesión en Twitter.

Figura 14. Twitter Application Management

En esta página podremos ver la lista de apps que tengamos creadas, y podemos crear nuevas aplicaciones pulsando el botón "*Create New App*". Solamente tendremos que rellenar los campos que se muestran a continuación.

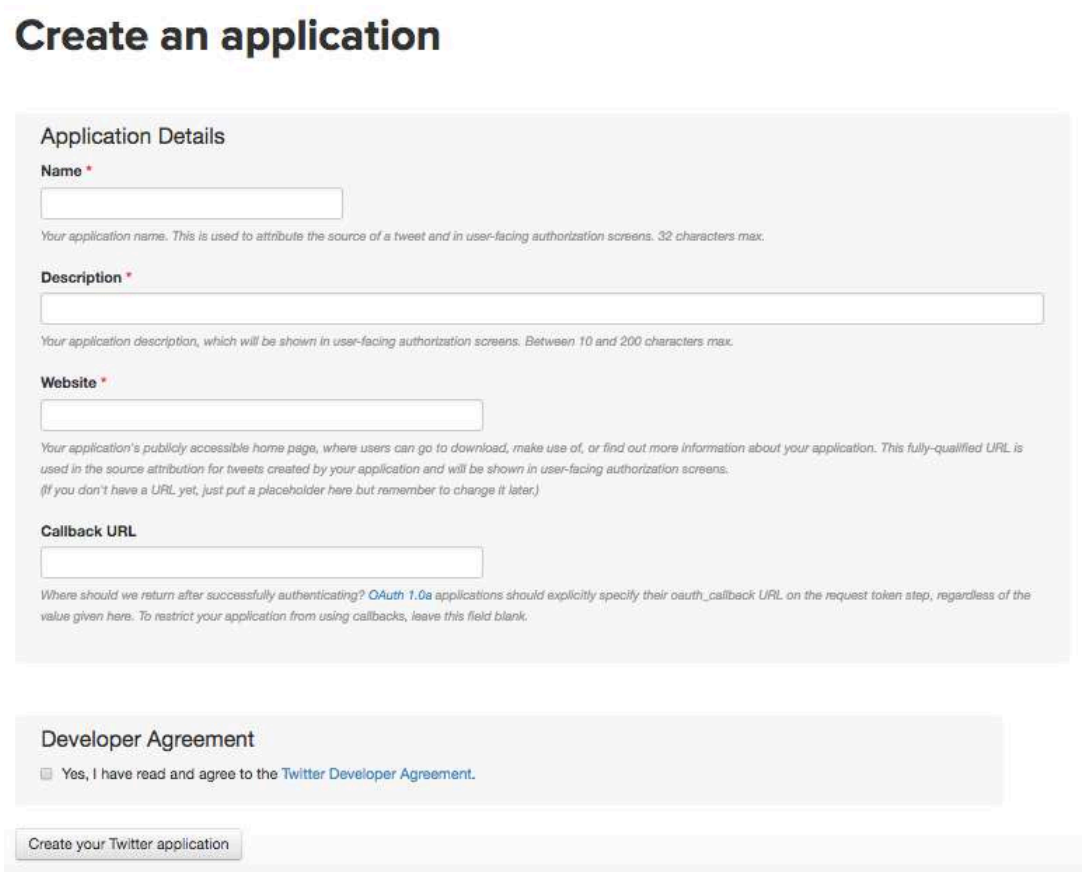

Figura 15. Crear aplicación en Twitter

Una vez registrada la aplicación, en la pestaña "***Key and Access Tokens***", podremos encontrar la información necesaria para autenticar nuestra aplicación. La ***Consumer Key*** y la ***Consumer Secret*** representan la aplicación.

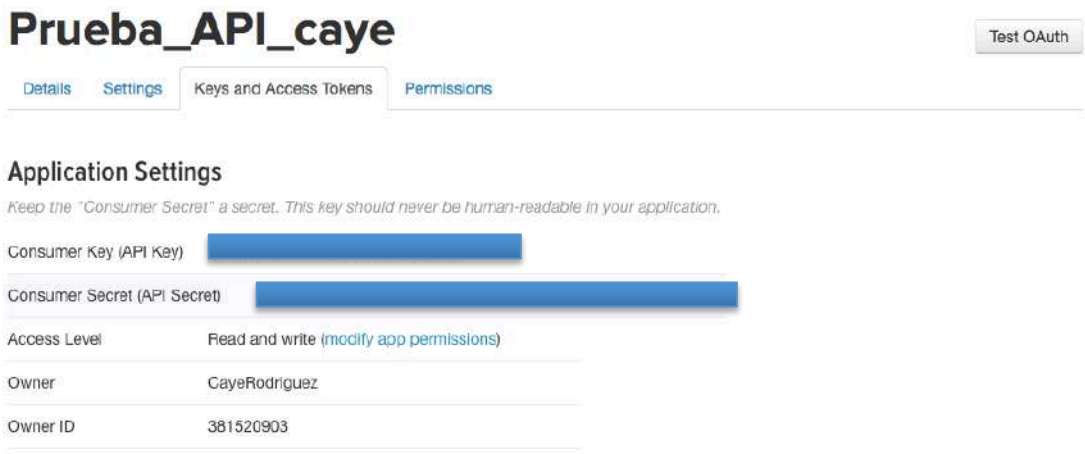

Figura 16. Consumer Key y Consumer Secret

Por otro lado, el ***Access Token*** y el ***Access Token Secret*** representan la cuenta del usuario.

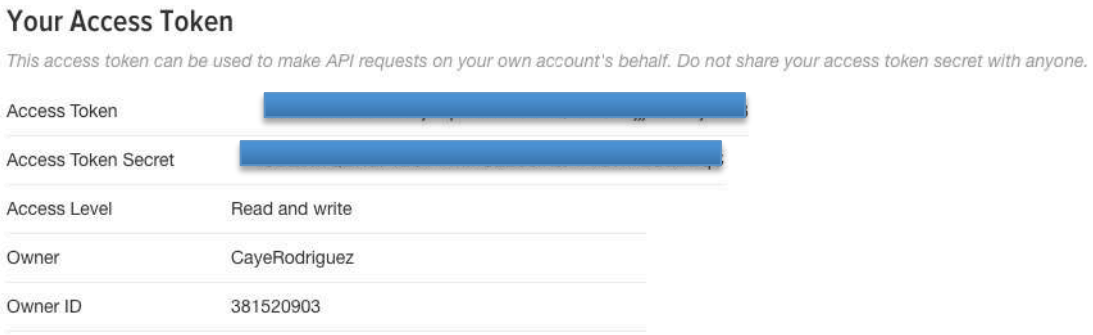

Figura 17. Access Token y Access Token Secret

Tendremos que usar estas cuatro claves en nuestro código para poder acceder a los datos de Twitter programáticamente.

3.2 APIs de Twitter

Twitter ofrece más que una única API. De hecho, hay distintas formas de acceder a los datos de Twitter. Con el fin de simplificar un poco la idea, podemos dividir estas opciones en dos clases: **APIs REST** y **Streaming API**.

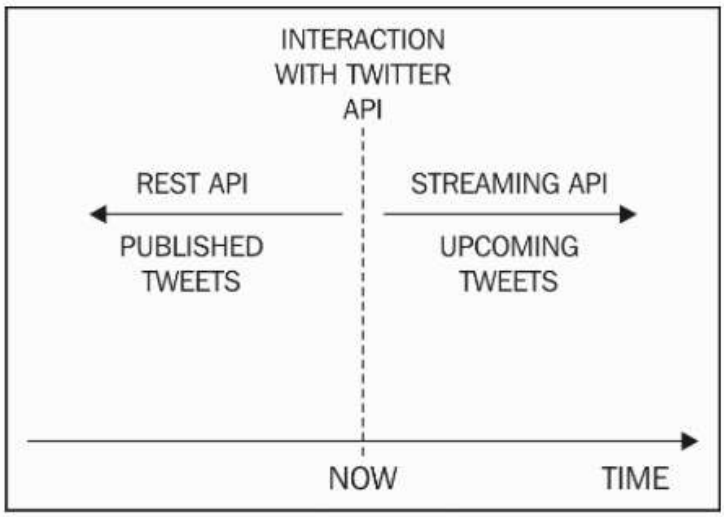

Figura 18. APIs de Twitter

La diferencia entre ellas es bastante simple: las APIs REST solo permiten ir atrás en el tiempo. Al interactuar con Twitter a través de una API REST, podemos buscar tweets ya existentes, es decir, tweets que ya han sido publicados y que están disponibles para su búsqueda. A menudo, este tipo de APIs tienen algunas limitaciones en cuanto al número de tweets que se pueden recuperar, así como de tiempo. Normalmente, es posible volver atrás en el tiempo hasta una semana, siendo tweets más antiguos imposibles de recuperar. Otro aspecto a considerar acerca de las APIs REST es que son *best-effort*, es decir, que no garantizan la devolución de todos los tweets publicados en Twitter.

Por otro lado, la Streaming API examina el futuro. Una vez abramos una conexión, podemos mantenerla abierta y avanzar en el tiempo. Manteniendo la conexión HTTP abierta, podemos recuperar todos los tweets que cumplan cierto criterio de filtrado, conforme se van publicando. Debido a esto, la Streaming API es, generalmente, la forma preferida de descargar una gran cantidad de tweets, ya que la interacción con la plataforma está limitada a mantener una conexión abierta. Como desventaja, este método suele consumir más tiempo, ya que tenemos que esperar a que los tweets se publiquen para poder recuperarlos.

Como resumen, las APIs REST son útiles cuando queremos buscar tweets de un usuario en concreto, o cuando queremos acceder a nuestro propio *timeline*, mientras que la Streaming API es más interesante cuando queremos filtrar por una palabra específica y descargar una cantidad masiva de tweets sobre ese tema (por ejemplo, eventos en directo).

3.3 Objetos

Existen cuatro "objetos" principales que podemos encontrar en la API de Twitter: ***Tweets***, ***Users***, ***Entities*** y ***Places***. Es fundamental conocer la anatomía de estos objetos para conocer sus propiedades y poder interactuar con ellos. Todos estos objetos son de tipo JSON, es decir, un conjunto de atributos del tipo par clave-valor.

3.3.1 Tweets

Los Tweets son el bloque atómico básico de todas las cosas en Twitter. También se conocen como "actualizaciones de estado". A continuación describiremos algunos de los campos más importantes contenidos en un tweet.

Tabla 1. Anatomía de un Tweet

Campo	Tipo	Descripción
id	`Int64`	Identificador único del tweet.
id_str	`String`	Representación del identificador en forma de *string*.
text	`String`	Contenido textual del mensaje de actualización de estado.
user	`Users`	Usuario que ha publicado el tweet.
created_at	`String`	Fecha de creación del tweet.
entities	`Entities`	Objetos de tipo `Entities` que han sido extraídos del tweet.
favorite_count	`Integer`	Número de veces que el tweet ha sido marcado como "me gusta" por un usuario.
favorited	`Boolean`	Indica si el tweet ha sido marcado como "me gusta" por algún usuario.
retweet_count	`Int`	Número de veces que el tweet ha sido retwitteado.
retweeted	`Boolean`	Indica si el tweet ha sido retwitteado por algún usuario.
coordinates	`Coordinates`	Localización geográfica, especificada por el usuario u otra aplicación. Aparece en formato *geoJSON* (longitud, latitud).
place	`Places`	Indica que el tweet está relacionado con (pero no necesariamente originado en) un lugar (objeto de tipo Place).

A modo de ejemplo se muestra el contenido de un tweet real con todos sus campos.

```
{

"created_at":"Thu Jun 30 23:26:12 +0000 2016",

"id":748658928867741696,

"id_str":"748658928867741696",

"text":"@Ismawel_mar estabas viendo la peli de telecinco o que? xD",

"truncated":false,

"entities":{

        "hashtags":[],

        "symbols":[],

        "user_mentions":[

                {

                        "screen_name":"Ismawel_mar",

                        "name":"Ismael Mart\u00edn",

                        "id":478619872,

                        "id_str":"478619872",

                        "indices":[0,12]

                }

        ],

        "urls":[]

},

"source":"<a href=\"http://twitter.com/download/iphone\"
rel=\"nofollow\">Twitter for iPhone</a>",

"in_reply_to_status_id":748657585222467584,

"in_reply_to_status_id_str":"748657585222467584",

"in_reply_to_user_id":478619872,

"in_reply_to_user_id_str":"478619872",

   "in_reply_to_screen_name":"Ismawel_mar",
```

```
"user":{
    "id":381520903,
    "id_str":"381520903",
    "name":"Cayetano Rodr\u00edguez",
    "screen_name":"CayeRodriguez",
    "location":"C\u00e1diz",
    "description":"Gaditano, Ingeniero en Tecnolog\u00edas de
    Telecomunicaci\u00f3n (Teleco pa' los amigos). Actualmente en
    Madrid",
    "url":"https://t.co/u0fosIFha5",
    "entities":{
        "url":{
            "urls":[
                {
                    "url":"https://t.co/u0fosIFha5",
                    "expanded_url":"http://www.youtube.com/us
                    er/cayetherenegade/featured",
                    "display_url":"youtube.com/user/cayethere
                    \u2026",
                    "indices":[0,23]
                }
            ]
        },
        "description":{
            "urls":[]
        }
    },
    "protected":false,
    "followers_count":126,
    "friends_count":246,
    "listed_count":0,
    "created_at":"Wed Sep 28 13:41:49 +0000 2011",
    "favourites_count":146,
```

```
        "utc_offset":7200,

           "time_zone":"Madrid",

        "geo_enabled":false,

        "verified":false,

        "statuses_count":6489,

        "lang":"es",

        "contributors_enabled":false,

        "is_translator":false,

        "is_translation_enabled":false,

        "profile_background_color":"022330",

        "profile_background_image_url":"http://pbs.twimg.com/profile_backgrou
        nd_images/543453382/Carnaval_2011.jpg",

        "profile_background_image_url_https":"https://pbs.twimg.com/profile_b
        ackground_images/543453382/Carnaval_2011.jpg",

        "profile_background_tile":true,

        "profile_image_url":"http://pbs.twimg.com/profile_images/578976365904
        556032/A_snK8Gd_normal.jpeg",

        "profile_image_url_https":"https://pbs.twimg.com/profile_images/57897
        6365904556032/A_snK8Gd_normal.jpeg",

        "profile_banner_url":"https://pbs.twimg.com/profile_banners/381520903
        /1426873699",

        "profile_link_color":"0084B4",

        "profile_sidebar_border_color":"A8C7F7",

        "profile_sidebar_fill_color":"C0DFEC",

        "profile_text_color":"333333",

        "profile_use_background_image":true,

        "has_extended_profile":false,

        "default_profile":false,

        "default_profile_image":false,

        "following":false,

        "follow_request_sent":false,

        "notifications":false,

        "translator_type":"none"

    },
```

```
"geo":null,
"coordinates":null,
"place":null,
"contributors":null,
"is_quote_status":false,
"retweet_count":0,
"favorite_count":1,
"favorited":false,
"retweeted":false,
"lang":"es"
```


Figura 19. Ejemplo de tweet

3.3.2 Users

Los objetos Users representan a los usuarios de Twitter, que pueden ser cualquier persona o cosa. Del mismo modo que con los tweets, describiremos los principales campos de un objeto de tipo User.

Tabla 2. Anatomía de un User

Campo	Tipo	Descripción
id	`Int64`	Identificador único del usuario.
id_str	`String`	Representación del identificador en forma de *string*.
created_at	`String`	Fecha de creación del perfil de usuario.
description	`String`	Descripción de la cuenta de usuario.
favourites_count	`Int`	Número de veces que el usuario ha marcado un tweet como ”me gusta”.
followers_count	`Int`	Número de seguidores de este usuario.
friends_count	`Int`	Número de usuarios que está siguiendo.
name	`String`	Nombre del usuario.
screen_name	`String`	Nombre en pantalla del usuario.
status	`Tweets`	Si es posible, último tweet publicado por el usuario.
statuses_count	`Int`	Número de tweets publicados por el usuario.

A modo de ejemplo se muestra el contenido de un tweet real con todos sus campos.


```
{
    "id":757965164423553025,
    "id_str":"757965164423553025",
    "name":"IngenieroJob",
    "screen_name":"IngenieroJob",
    "location":"contact@ingenierojob.com",
    "description":"TALENT MOVES THE WORLD \u2022 El primer portal de empleo de
    ingenier\u00eda de habla hispana \u2022 Si eres ingeniero y buscas trabajo,
    \u00a1\u00danete!",
    "url":"https://t.co/7pUzdFN5dU",
    "entities":{
        "url":{
            "urls":[
                {
                    "url":"https://t.co/7pUzdFN5dU",
                    "expanded_url":"http://ingenierojob.com",
                    "display_url":"ingenierojob.com",
                    "indices":[0,23]
                }
            ]
        },
        "description":{
            "urls":[]
        }
    },
    "protected":false,
    "followers_count":2465,
    "friends_count":1,
```

```
"listed_count":3,
"created_at":"Tue Jul 26 15:45:52 +0000 2016",
"favourites_count":85,
"utc_offset":3600,
"time_zone":"London",
"geo_enabled":false,
"verified":false,
"statuses_count":101,
"lang":"es",
"status":{
    "created_at":"Thu Apr 20 09:42:29 +0000 2017",
    "id":854993666821238784,
    "id_str":"854993666821238784",
    "text":"Ingenieros, hay nuevas ofertas de trabajo en
    https://t.co/g3reypG8OL\n\u00a1Ya no ten\u00e9is excusas!",
    "truncated":false,
    "entities":{
        "hashtags":[],
        "symbols":[],
        "user_mentions":[],
        "urls":[
            {
                "url":"https://t.co/g3reypG8OL",
                "expanded_url":"http://ingenierojob.com",
                "display_url":"ingenierojob.com",
                "indices":[45,68]
            }
        ]
    },
```

```
"source":"<a href=\"http://twitter.com\" rel=\"nofollow\">Twitter Web
Client</a>",

"in_reply_to_status_id":null,

"in_reply_to_status_id_str":null,

"in_reply_to_user_id":null,

"in_reply_to_user_id_str":null,

"in_reply_to_screen_name":null,

"geo":null,

"coordinates":null,

"place":null,

"contributors":null,

"is_quote_status":false,

"retweet_count":5,

"favorite_count":2,

"favorited":false,

"retweeted":false,

"possibly_sensitive":true,

"lang":"es"
```

```
},

"contributors_enabled":false,

"is_translator":false,

"is_translation_enabled":false,

"profile_background_color":"000000",

"profile_background_image_url":"http://abs.twimg.com/images/themes/theme1/bg.
png",

"profile_background_image_url_https":"https://abs.twimg.com/images/themes/the
me1/bg.png",

"profile_background_tile":false,

"profile_image_url":"http://pbs.twimg.com/profile_images/846434824185401344/M
Jbg-y0H_normal.jpg",

   "profile_image_url_https":"https://pbs.twimg.com/profile_images/8464348241
85401344/MJbg-y0H_normal.jpg",
```

```
"profile_banner_url":"https://pbs.twimg.com/profile_banners/757965164423553025/1491205685",

"profile_link_color":"19CF86",

"profile_sidebar_border_color":"000000",

"profile_sidebar_fill_color":"000000",

"profile_text_color":"000000",

"profile_use_background_image":false,

"has_extended_profile":false,

"default_profile":false,

"default_profile_image":false,

"following":true,

"follow_request_sent":false,

"notifications":false,

"translator_type":"none"

}
```


Figura 20. Ejemplo de User

3.3.3 Entities

Los objetos de tipo Entity proporcionan metadatos e información de contexto adicional sobre el contenido publicado en Twitter. Las más importantes son:

- **Hashtags**: etiquetas. Palabra o conjunto de palabras precedidos por el símbolo de la almohadilla (#).
- **Media**: imágenes o vídeos.
- **URLs**: enlaces a direcciones de Internet (http://....).
- **Menciones a usuarios**: en Twitter es posible enviar un tweet mencionando a otro usuario. Esto se hace utilizando el nombre de dicho usuario precedido del símbolo @.

3.3.4 Places

Se trata de localizaciones específicas con unas coordenadas geográficas. Es posible unirlos a un tweet especificando un `place_id` al publicar el tweet. Los tweets asociados con un lugar no tienen por qué ser emitidos desde ese lugar, sino que podrían comentar algo sobre dicho lugar. No entraremos en más detalles sobre este tipo de objetos debido a que no se usarán en este proyecto.

3.4 Rate limits

La API de Twitter limita el acceso a sus aplicaciones. Estos límites se establecen por usuario, o más específicamente, por token de acceso. Cada API tiene un límite de uso distinto, y es importante conocerlos para no sobrepasarlos. En el caso de que sobrepasemos algún límite de uso, Twitter devolverá un error en lugar de la respuesta a lo que se haya pedido. Es más, si continuamos realizando peticiones a la API, el tiempo necesario para obtener acceso regular de nuevo, irá incrementando.

Los rate limits en la Streaming API funcionan de forma diferente al resto de APIs. En este caso no se limita el número de peticiones, ya que al ser en streaming no se van realizando peticiones continuas, sino que se establece una conexión en el momento inicial y se van recibiendo los tweets que cumplan con los criterios especificados. Sin embargo, los clientes que intenten reconectarse tan rápido como puedan sin implementar un backoff, serán castigados limitando el número de conexiones durante varios minutos. Si pasa esto, se recibirán mensajes HTTP 420 a todas las peticiones de conexión que realicen. Por tanto, los clientes que rompan una conexión y vuelvan a reconectarse frecuentemente (para cambiar los parámetros de las consultas, por ejemplo), corren el riesgo de verse en esta situación. Si Twitter considera que un usuario hace esto demasiadas veces, puede decidir bloquear su IP para acceder a Twitter durante un intervalo de tiempo indeterminado.

4 Implementación de la plataforma de procesamiento de datos

Se va a implementar una plataforma de procesamiento de datos, donde se analizará la información proveniente de varias redes sociales y de la web 2.0. Para ello se hará uso de técnicas de Big Data y se emplearán las soluciones tecnológicas de Apache Spark. Sobre una infraestructura de computación en la nube se desplegará una plataforma de procesamiento donde se combinen distintas tecnologías como Jupyter, Apache Kafka o Apache ZooKeeper.

4.1 Infraestructura de computación en la nube: Amazon Web Services

Desde un primer momento, el objetivo ha sido crear un escenario lo más real posible. Para conseguirlo, se optó por construir la plataforma de procesamiento de datos sobre una infraestructura de computación en la nube, en la que se montará un cluster de Apache Spark con múltiples nodos, ampliables de una manera muy sencilla gracias a las facilidades de Amazon Web Services. Por tanto, la plataforma de procesamiento de datos completa se construirá en la nube, haciendo uso de las instancias EC2 de AWS. A continuación se detallarán los pasos necesarios para crear y configurar una instancia EC2 en Amazon Web Services, lo cual será necesario hacer varias veces a lo largo del proyecto.

4.1.1 Lanzamiento de una instancia EC2

Antes que nada, es necesario tener una cuenta de Amazon Web Services e iniciar sesión. Al identificarnos en la página web de Amazon Web Services (aws.amazon.com), accederemos a nuestro dashboard de AWS. Si pulsamos sobre "*Services*" podremos ver todos los servicios ofrecidos por AWS. Aquí debemos seleccionar EC2.

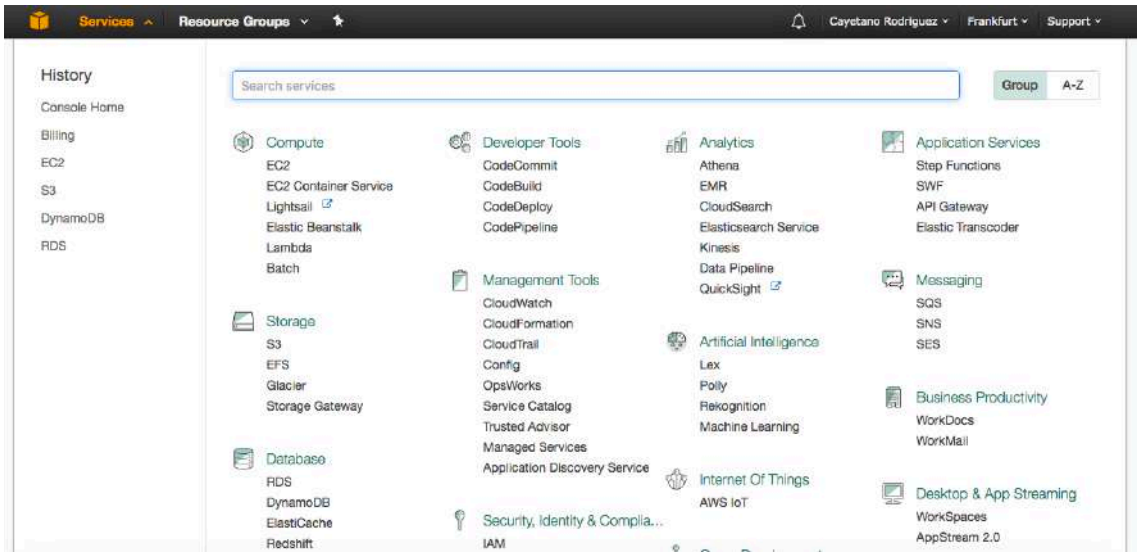

Figura 21. Servicios disponibles en el dashboard de AWS

A continuación, veremos el dashboard de EC2, donde podremos ver una vista general del uso de este servicio.

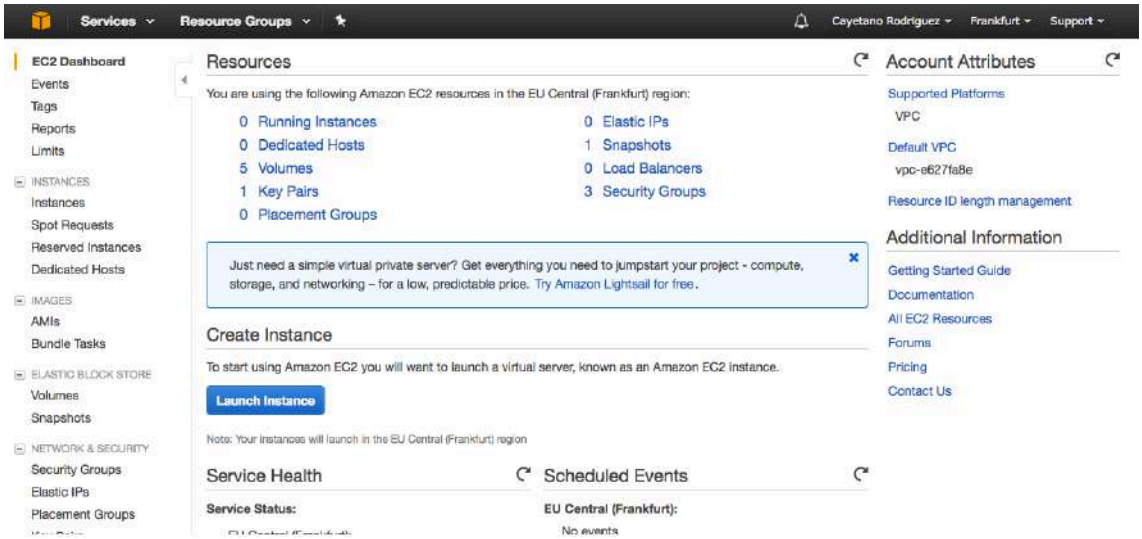

Figura 22. Dashboard de EC2

Si pulsamos sobre "*Instances*" podremos ver las instancias que hayamos creado, así como modificar su estado o conectarnos a ellas, o bien, lanzar una nueva instancia. Esto último es lo que tendremos que hacer. Pulsamos sobre "*Launch Instance*".

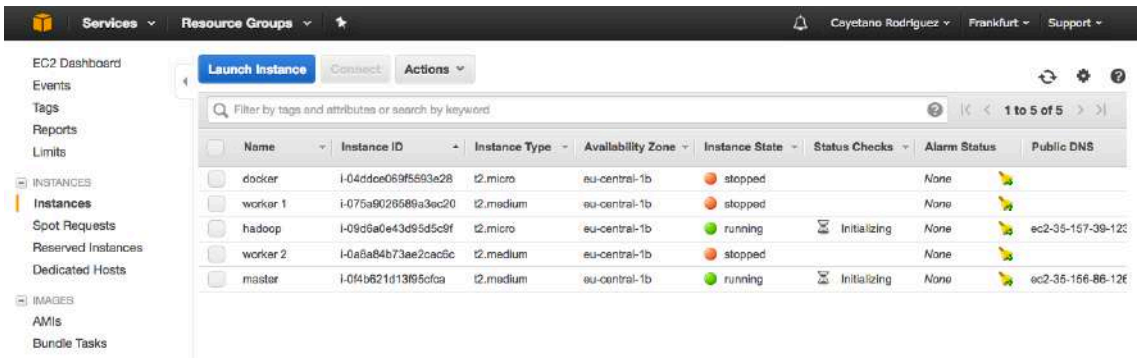

Figura 23. Listado de instancias EC2

A continuación, se abrirá una nueva ventana, en la que AWS nos irá guiando paso a paso para que configuremos correctamente nuestra instancia.

4.1.2 Configuración de la instancia

El primer paso será seleccionar la imagen (AMI) que usaremos para nuestra instancia. La imagen que utilizaremos para los nodos de Apache Spark será "Ubuntu Server 16.04".

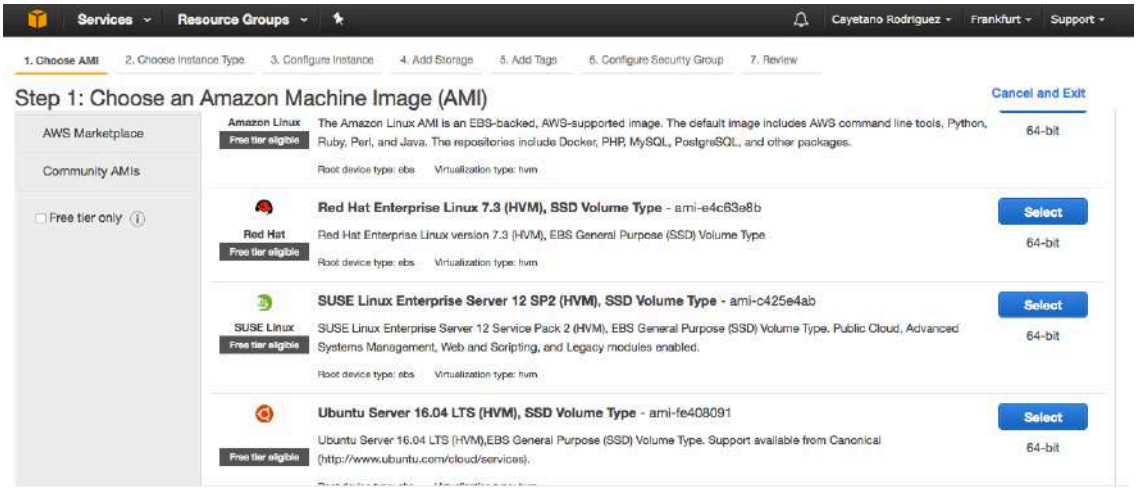

Figura 24. Configuración de la instancia: AMI

El segundo paso consiste en seleccionar el tipo de instancia. La única que es gratuita es la t2.micro, que consta de 1 CPU y 1 GB de memoria. Para la instancia en la que desplegaremos el máster y los workers de Spark, hará falta algo mejor, por lo que usaremos, como mínimo, una t2.medium, con 2 CPUs y 4 GB de memoria.

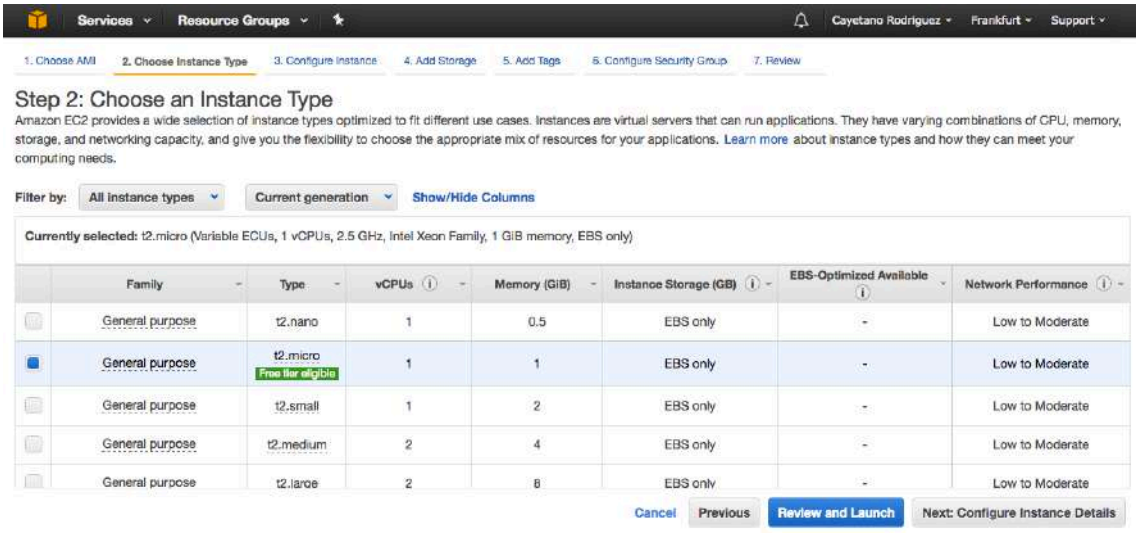

Figura 25. Configuración de la instancia: tipo de instancia

A continuación, tendremos que especificar la configuración básica de la instancia. Aquí podremos especificar si queremos lanzar una única instancia o varias, con la misma configuración. El resto de parámetros pueden dejarse con el valor por defecto.

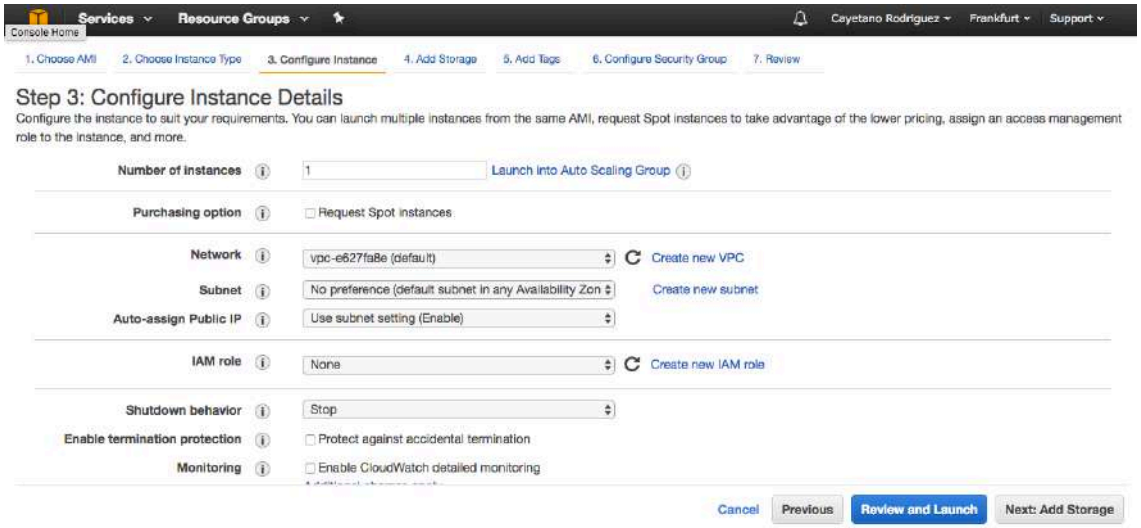

Figura 26. Configuración de la instancia: detalles de la instancia

El siguiente paso será añadir el almacenamiento para la instancia. Elegiremos un volumen de 8 GB de tipo GP2, para una mayor velocidad.

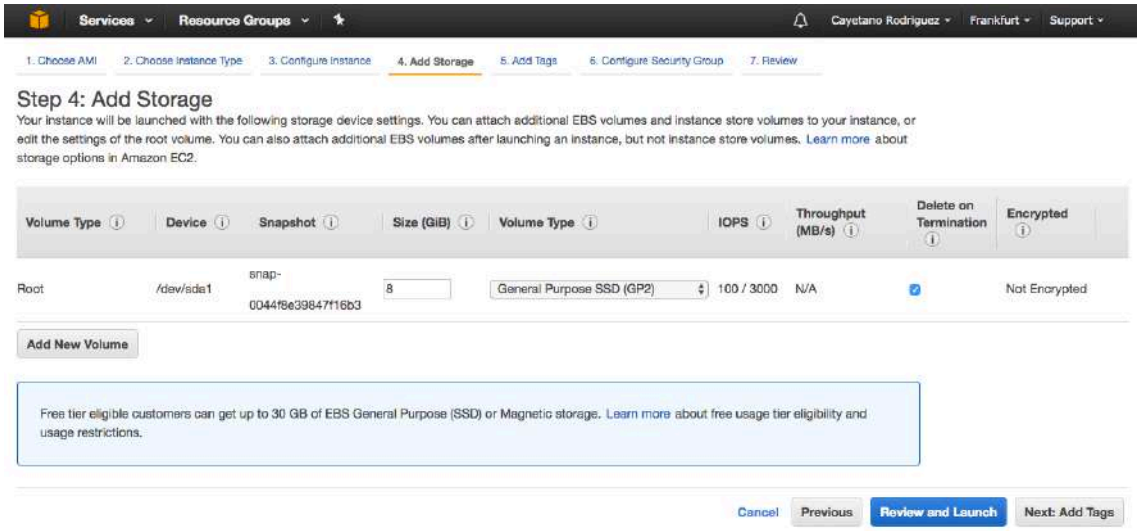

Figura 27. Configuración de la instancia: almacenamiento

Posteriormente, podremos especificar un nombre para la instancia, de modo que podamos identificarla fácilmente.

Figura 28. Configuración de la instancia: etiquetas

El último paso será configurar los grupos de seguridad que se aplicarán a nuestra instancia. Podemos seleccionar uno por defecto, o bien, crear uno nuevo, abriendo los puertos que creamos conveniente. Los grupos de seguridad pueden modificarse más adelante, con lo cual no es algo demasiado importante ahora mismo.

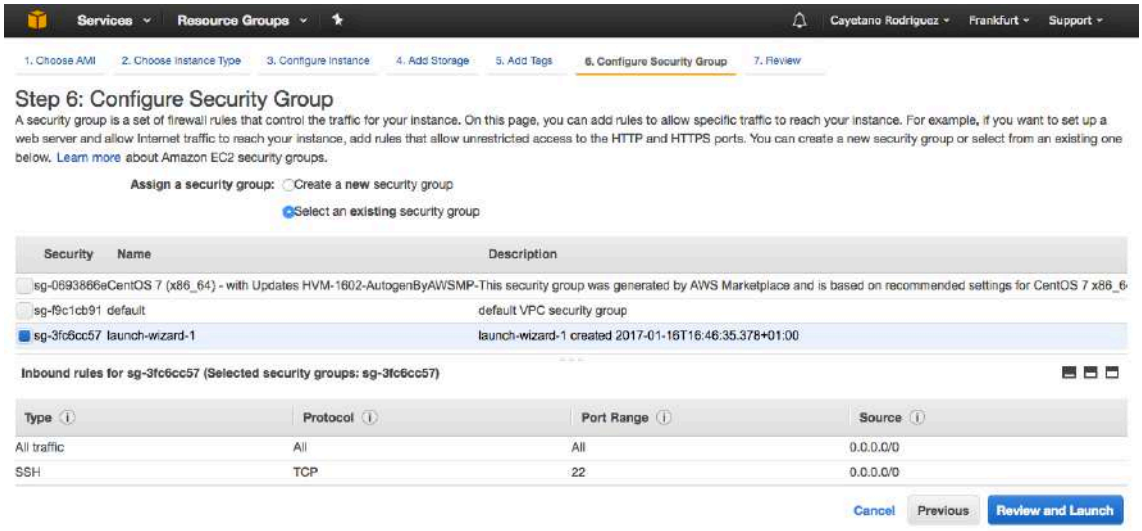

Figura 29. Configuración de la instancia: grupos de seguridad

Finalmente, podemos pulsar sobre "*Review and Launch*", donde nos aparecerá un resumen de la configuración que hemos especificado. Si todo está correcto, pulsamos sobre "*Launch*". En este momento, aparecerá una ventana emergente que nos pedirá una *key pair*. Se trata de una pareja de claves (pública y privada), que tendremos que optar entre crear una nueva o utilizar una antigua. Si decidimos crear una nueva, tendremos que descargar la clave privada, y la clave pública será almacenada por AWS, y será lo que utilice para permitirnos acceder a nuestra instancia de forma segura mediante SSH.

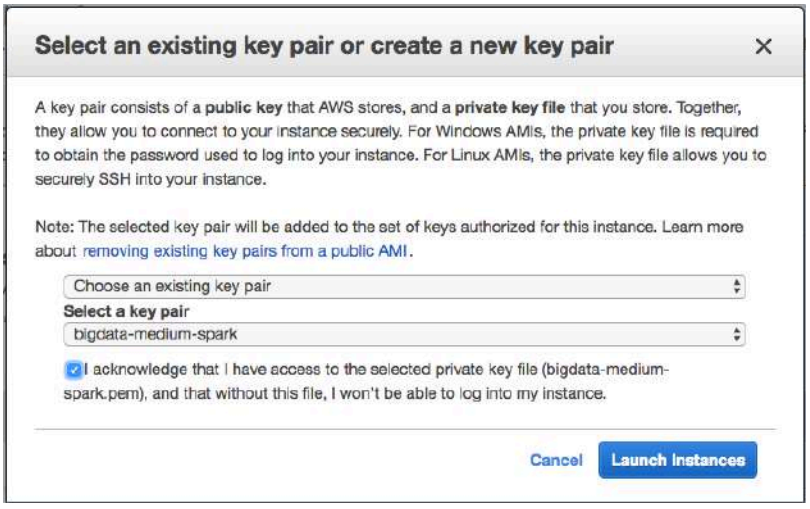

Figura 30. Configuración de la instancia: key pair

4.1.3 Conexión con la instancia

Conectarse a la instancia es realmente sencillo. Lo haremos mediante SSH y utilizando la clave privada que hemos descargado previamente. En el dashboard de EC2, podemos pulsar sobre la instancia a la que queremos conectarnos, y pulsar sobre "*Connect*". Se abrirá una ventana emergente con las instrucciones que nos permitirán realizar la conexión de una forma muy sencilla.

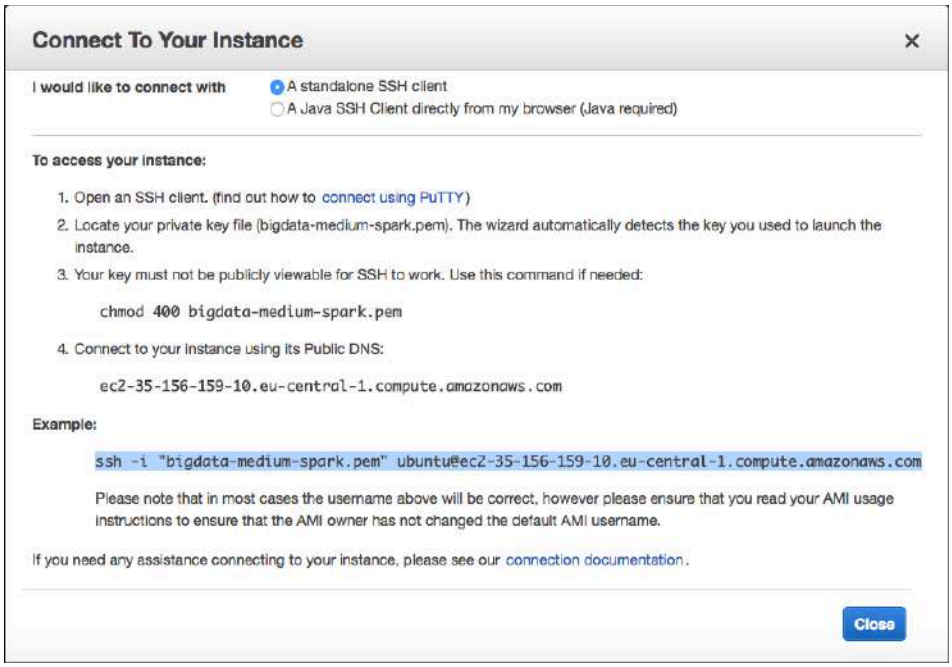

Figura 31. Conexión con la instancia

Tendremos que modificar los permisos de la clave privada:

```
chmod 400 clave-privada.pem
```

Nos bastaría con copiar la línea que aparece marcada en la imagen anterior con el comando `ssh` necesario para realizar la conexión, que tiene la siguiente forma:

```
ssh -i "clave-privada.pem" usuario@ec2-ip-de-la-instancia.región-de- la-instancia-compute.amazonaws.com
```

4.1.4 Modificar el estado de una instancia

Otro aspecto de interés a la hora de trabajar con instancias EC2 es saber cómo modificar el estado de una instancia. Esto nos permitirá arrancar, parar, reiniciar o terminar una instancia definitivamente. Para hacerlo, simplemente tenemos que acceder al listado de instancias EC2, pulsar sobre la instancia en cuestión y posteriormente pulsar sobre *Actions* → *Instance State* y seleccionar el estado al que queremos que pase la instancia (*Start, Stop, Reboot* o *Terminate*).

4.2 Instalación de Jupyter y Anaconda

Para instalar Jupyter, en primer lugar instalaremos Anaconda Python, el cual incluye otros paquetes de interés y ofrece facilidades para realizar la instalación de Jupyter con un solo comando.

4.2.1 Prerrequisitos y especificaciones

- Licencia gratuita.
- Sistema operativo: Windows Vista o en adelante, Mac OS 10.7+, o Linux (Ubuntu, CentOS 5+…).
- 32-bit o 64-bit.
- Mínimo 3 GB de espacio en disco para descargar e instalar.

4.2.2 Instalar Anaconda

La instalación de Anaconda es muy sencilla, tan solo hay que acceder a la página de descargas oficial https://www.continuum.io/downloads, seleccionar el sistema operativo (en nuestro caso, Linux) y escoger la versión de Python que queremos instalar. Nos interesa utilizar Python 3, aunque la versión 3.6 de Python (la última versión disponible de Python y que viene en la última versión de Anaconda 4.3.0) no es compatible con Apache Spark 2.1.0. Por tanto, tendremos que buscar la última versión de Anaconda que venga con Python 3.5. Para ello, tenemos que acceder al "*Anaconda installer archive*" https://repo.continuum.io/archive/index.html, donde se almacenan las versiones anteriores de Anaconda, y buscar la que nos interesa. Tenemos que asegurarnos de seleccionar la versión de Anaconda 3 anterior a la actual, de modo que utilice Python 3.5, que será la 4.2.0, para Linux y 64-bits. El fichero en cuestión es ***Anaconda3-4.2.0-Linux-x86_64.sh***, que podemos buscar en la lista de ficheros de instalación del "*Anaconda installer archive*", y cuya dirección de descarga es la siguiente: https://repo.continuum.io/archive/Anaconda3-4.2.0-Linux-x86_64.sh.

Tendremos que realizar la instalación en la instancia de EC2 donde se situará el nodo master de Apache Spark, de modo que Jupyter se encuentre en la misma máquina que el master de Spark.

Para realizar la instalación, nos conectamos a la instancia en cuestión y ejecutamos el siguiente comando:

```
curl -O dirección_descarga_Anaconda3-4.2.0
```

Se nos descargará un script de instalación, al cual hay que darle permiso de ejecución:

```
chmod +x Anaconda3-4.2.0-Linux-x86_64.sh
```

Tras esto, ya podremos ejecutarlo. Al hacerlo, se realizarán una serie de cuestiones, que nos guiarán a lo largo de la instalación. Al finalizar, nos pregunta si queremos añadir la dirección de instalación de Anaconda (`/home/ubuntu/anaconda3`) al `PATH`. Lo que hace es añadir una línea al fichero `.bashrc` exportando la variable `PATH` con el nuevo directorio. Para que se haga efectivo el cambio, tendremos que reiniciar la instancia, o bien ejecutar el comando:

```
source .bashrc
```

Con esto, ya tenemos Anaconda instalado en nuestra instancia, y podemos proceder con la instalación de Jupyter.

4.2.3 Instalar Jupyter

La instalación de Jupyter se puede hacer fácilmente utilizando el gestor de paquetes ***conda***, que viene con Anaconda. Nos basta con ejecutar:

```
conda install jupyter
```

Al haber añadido Anaconda al `PATH`, el comando anterior debería funcionar perfectamente. En caso de que falle, nos bastaría con situarnos en el directorio donde se encuentra el ejecutable de conda (`/anaconda3/bin`) y volver a ejecutar el comando anterior.

Tras unos segundos, ya se habrá instalado Jupyter y podremos ejecutarlo, para poder crear notebooks.

```
jupyter notebook --ip=dirección_pública_AWS
```

Es importante especificar el parámetro `--ip` con la dirección pública de Amazon Web Services de la instancia. Por defecto, Jupyter se ejecuta de forma local, por tanto, para poder utilizar Jupyter en la instancia EC2 tenemos que ejecutarlo usando la dirección pública, de modo que se pueda acceder a la interfaz web.

Para comprobar que Jupyter está corriendo correctamente, accedemos a la interfaz web, haciendo uso de la dirección pública de AWS y el puerto 8888. La primera vez que accedamos a la interfaz web, tendremos que usar un token, que nos aparecerá en la terminal de comandos al ejecutar el comando anterior. Posteriormente, podremos acceder a la interfaz directamente a través de la siguiente dirección:

```
https://dirección_pública_AWS:8888/
```

Debemos ver una interfaz como la de la siguiente imagen.

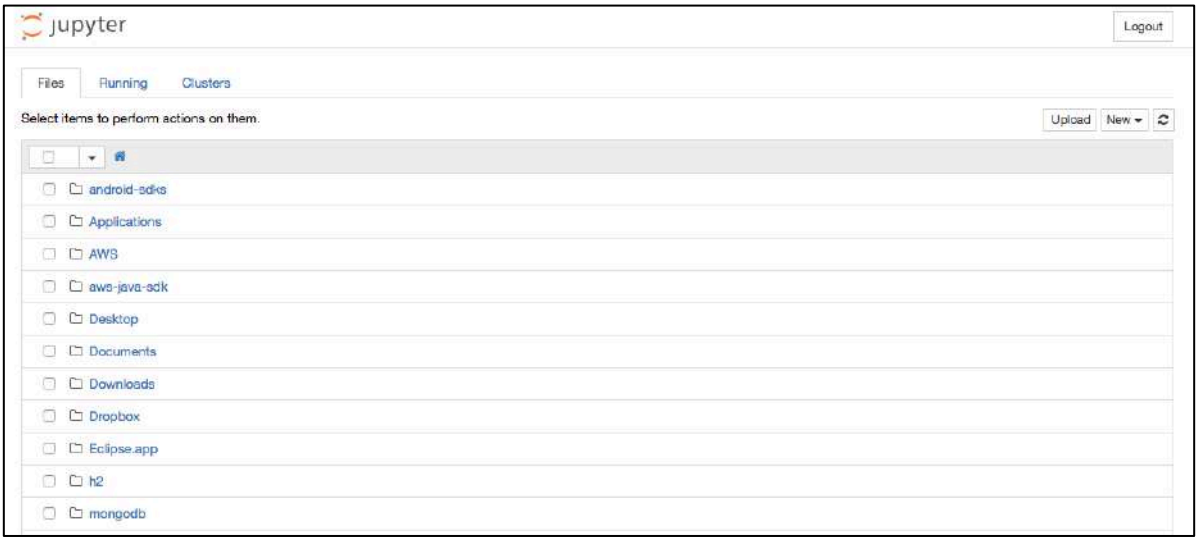

Figura 32. Interfaz web Jupyter

4.2.4 Configuración de Jupyter

Para poder usar Jupyter sin problemas en la instancia de Amazon, y conjuntamente con Apache Spark, es necesario realizar una pequeña configuración adicional.

En primer lugar, tendremos que crear el fichero de configuración con el comando:

```
jupyter notebook --generate-config
```

A continuación, crearemos los certificados para poder conectarnos a la interfaz de forma segura, mediante certificados:

```
mkdir certs

cd certs

sudo openssl req -x509 -nodes -days 365 -newkey rsa:1024 -keyout mycert.pem -out mycert.pem
```

Nos harán varias preguntas generales, que podemos responder o dejar sin respuesta. Posteriormente, editaremos el fichero de configuración que generamos antes.

```
cd ~/.jupyter/

sudo vim jupyter_notebook_config.py
```

Al comienzo del fichero, añadimos el siguiente trozo de código:

```
c = get_config()

# Notebook config this is where you saved your pem cert
c.NotebookApp.certfile = u'/home/ubuntu/certs/mycert.pem'

# Run on all IP addresses of your instance
c.NotebookApp.ip = '*'

# Don't open browser by default
c.NotebookApp.open_browser = False

# Fix port to 8888
c.NotebookApp.port = 8888
```

Con esto conseguimos lo siguiente:

- Hacemos que se ejecute en todas las direcciones IP de la instancia. Con esto evitamos tener que usar el parámetro `--ip` al arrancarlo.
- Hacemos que no intente abrir un navegador por defecto, ya que al estar en una instancia de Amazon, no dispone de navegador.
- Fijamos el puerto en 8888 (podría usarse otro puerto distinto).

Finalmente, comprobaremos que todo sigue funcionando sin problemas, ejecutando de nuevo:

```
jupyter notebook
```

Si ahora accedemos a la interfaz web mediante:

https://dirección_pública_AWS:8888/

Es importante utilizar HTTPS en lugar de HTTP. Cuando intentemos acceder, tendremos que aceptar la conexión, ya que es posible que aparezca una advertencia de seguridad (debido al uso de certificados). Una vez hecho esto, podremos acceder de nuevo a la interfaz web de Jupyter.

4.3 Instalación de Apache Spark

Podemos proceder a instalar Apache Spark, componente principal de la plataforma de procesamiento de datos.

4.3.1 Prerrequisitos

Para instalar Apache Spark, es necesario instalar otros paquetes previamente. En primer lugar, ejecutamos:

```
sudo apt-get update
```

Necesitamos instalar Java:

```
sudo apt-get install openjdk-8-jdk
```

Debemos configurar la variable de entorno JAVA_HOME, de modo que Java sea localizable en el sistema. Para ello, añadiremos al fichero .bashrc la siguiente línea:

```
export JAVA_HOME="/usr/lib/jvm/java-1.8.0-openjdk-amd64"
```

Para que este cambio se haga efectivo, debemos reiniciar el sistema, o bien, ejecutar:

```
source .bashrc
```

También necesitaremos Scala:

```
sudo apt-get install scala
```

Por último, necesitamos instalar la librería de Python py4j. Para ello, tenemos que asegurarnos de tener instalado pip y conectado a Anaconda.

```
conda install pip

pip install py4j
```

4.3.2 Instalar Apache Spark

El primer paso será ir a la página de descargas oficial de Apache Spark (http://spark.apache.org/downloads) y descargar la última versión estable, la 2.1.0. También tenemos que seleccionar el tipo de paquete para el que queremos Spark, en nuestro caso será "*Pre-built for Hadoop 2.7 and later*". Copiamos el enlace de descarga de la página (http://d3kbcqa49mib13.cloudfront.net/spark-2.1.0-bin-hadoop2.7.tgz) y ejecutamos el siguiente comando:

```
curl –O dirección_descarga_Spark-2.1.0
```

Esto nos descargará un fichero comprimido en formato `tar.gz`, que podremos descomprimir de la siguiente forma:

```
tar -xvzf spark-2.1.0-bin-hadoop2.7.tgz
```

Así, se nos creará un directorio con todo lo necesario para que funcione Apache Spark en su interior. Ya podemos comprobar su correcto funcionamiento.

Para ejecutar un nodo master:

```
sbin/start-master.sh
```

Para ejecutar un nodo worker:

```
sbin/start-slave.sh spark://dirección_IP_master:7077
```

A los workers hay que especificarles en qué dirección está el nodo master al que se tienen que conectar. En nuestro caso, al estar en AWS, tendremos que especificar la dirección IP pública de la instancia en la que se encuentre el master.

Finalmente, para corroborar el correcto funcionamiento, podemos acceder a la interfaz web del master en la siguiente dirección:

```
http://dirección_IP_master:8080
```

Ahí, debemos ver algo como la siguiente imagen.

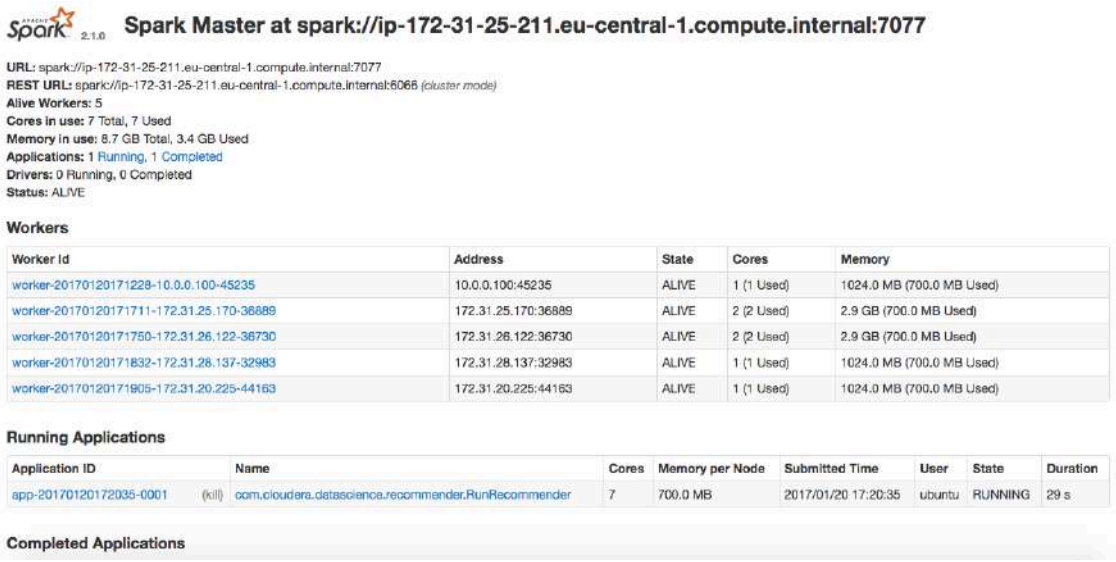

Figura 33. Interfaz web del master de Apache Spark

Si podemos acceder a esta dirección significa que el master se encuentra corriendo y funcionando correctamente. Además, en esta misma página podremos ver los workers que están actualmente conectados al master. Si se ejecuta una aplicación en Spark, también aparecerá en la interfaz web, junto con el número de cores y la memoria asignada para la aplicación.

También es interesante destacar que es posible arrancar y parar los workers en medio de la ejecución de una aplicación (siempre y cuando quede al menos un worker ejecutándola), y se seguirá ejecutando sin problemas, con lo cual, si un nodo falla, o si necesitamos añadir más workers para acelerar la ejecución de una aplicación, podemos hacerlo en tiempo de ejecución sin ningún problema.

4.4 PySpark

PySpark es el intérprete de Python incluido en Apache Spark. Desde una shell permite ejecutar comandos y aplicaciones Python en el cluster de Spark. Para ejecutar PySpark, tendremos que situarnos en el directorio de Apache Spark y ejecutar el comando:

```
./bin/pyspark --master spark://dirección_IP_master:7077
```

Al hacer esto, veremos cómo se abre la shell de PySpark y podremos ejecutar comandos Python en nuestro cluster de Apache Spark. Si accedemos de nuevo a la interfaz web del master de Spark, podremos ver que se añade una nueva aplicación con el nombre ***PySparkShell***.

Es importante destacar que el PySpark que viene con la versión 2.1.0 de Apache Spark no es compatible con Python 3.6.0, lo cual será resuelto en la próxima release de Spark. Por este motivo, instalamos la versión 4.2.0 de Anaconda, que viene con Python 3.5.

4.5 Enlazar PySpark y Jupyter

Nuestro objetivo es utilizar Jupyter para ejecutar las aplicaciones Python en el cluster de Apache Spark. Para ello, tendremos que enlazar PySpark, que es el intérprete de Python de Spark, con Jupyter.

El primer paso será decirle a Python dónde está Spark. Añadiremos las siguientes líneas al fichero `.bashrc`:

```
export SPARK_HOME="/home/ubuntu/spark-2.1.0-bin-hadoop2.7"

export PATH=$SPARK_HOME/bin:$PATH

export PYTHONPATH=$SPARK_HOME/python:$PYTHONPATH
```

Con esto, añadimos la variable de entorno `SPARK_HOME` al `PATH` y al `PYTHONPATH`, de modo que será accesible por Python. De nuevo, para hacer efectivos los cambios en el fichero `.bashrc`, habrá que reiniciar el sistema, o bien, ejecutar:

```
source .bashrc
```

Por otro lado, tendremos que añadir otras variables de entorno para que se ejecute PySpark conjuntamente con Jupyter. De nuevo, en el fichero `.bashrc`:

```
export PYSPARK_DRIVER_PYTHON=jupyter

export PYSPARK_DRIVER_PYTHON_OPTS='notebook' pyspark
```

Nuevamente, tendremos que hacer efectivos los cambios realizados en este fichero.

Si ahora ejecutamos:

```
jupyter notebook
```

Debemos ser capaces de acceder a la interfaz web de Jupyter. Si creamos un nuevo notebook usando el Kernel *python (conda)* y añadimos al comienzo de este fichero las siguientes líneas de código:

```
from pyspark import SparkContext, SparkConf

conf = SparkConf().setAppName("nombre_app")

conf.setMaster("spark://dirección_IP_master:7077")

sc = SparkContext.getOrCreate(conf=conf)
```

Con estas líneas, importaremos el contexto de Spark y lo configuraremos para especificar un nombre de aplicación, y el máster de Spark de nuestro cluster. Si todo ha ido bien, ya estaremos ejecutando los siguientes comandos que escribamos en el notebook en el cluster de Spark. Para comprobar que ha funcionado, si vamos a la interfaz web del master de Spark podremos ver una nueva aplicación en ejecución con el nombre que hayamos especificado en la configuración.

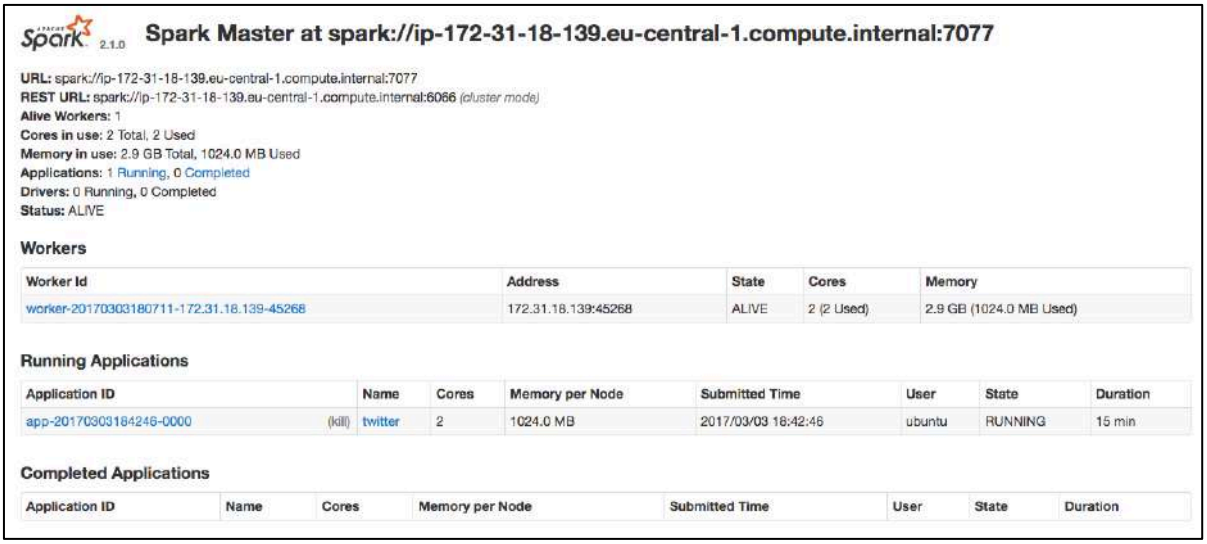

Figura 34. Interfaz web de Spark con aplicación ejecutada desde Jupyter

4.6 Esquema básico

A modo de resumen, la plataforma de procesamiento de datos que hemos diseñado e implementado ha quedado de la siguiente forma: tenemos una máquina principal que contiene el nodo máster de Spark y el servicio Jupyter (instalado sobre Anaconda), donde se ejecutarán las aplicaciones; por otro lado, tenemos otras n máquinas que servirán como nodos workers de Spark. Estos workers se encargarán de conectarse con el nodo máster para recibir las órdenes de ejecución oportunas. Cada una de estas máquinas es una instancia EC2 en la nube de Amazon Web Services

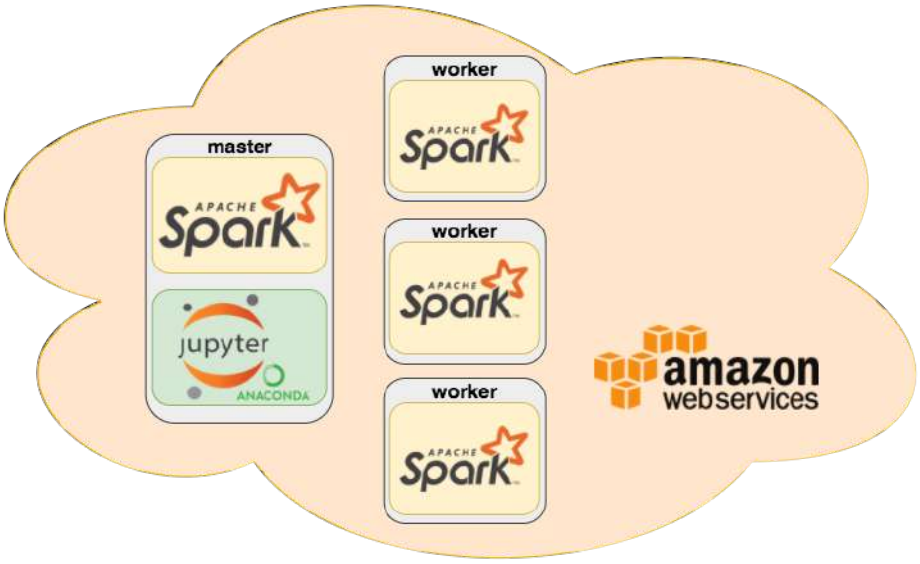

Figura 35. Escenario básico de plataforma de procesamiento de datos

5 Introducción de mejoras a la plataforma de procesamiento de datos

En el capítulo anterior hemos desarrollado una plataforma de procesamiento de datos básica. Sin embargo, es posible introducir una serie de mejoras técnicas que harán de esta plataforma una solución más completa, funcional y escalable. Estas mejoras incluyen: un sistema de almacenamiento distribuido, automatización y uso de un DNS dinámico para los nombres de las máquinas importantes.

5.1 HDFS

HDFS es el sistema de ficheros distribuido de Hadoop, y es altamente tolerante a fallos. Vamos a utilizarlo como sistema de almacenamiento distribuido para nuestro escenario completo.

HDFS tiene una arquitectura *master/slave* (maestro/esclavo). Un clúster HDFS consta de un único *NameNode*, un servidor máster que gestiona el *namespace* del sistema de ficheros y controla el acceso a los ficheros por parte de los clientes. Además, consta de un cierto número de *DataNodes*, normalmente uno por nodo en el clúster, que gestionan el almacenamiento correspondiente a los nodos sobre los que corren. HDFS expone un *namespace* de un sistema de ficheros y permite que los datos de los usuarios se almacenen en ficheros. Internamente, un fichero se divide en bloques, que se almacenan en un conjunto de *DataNodes*. Los *DataNodes* son los responsables de servir las peticiones de lectura y escritura del sistema de ficheros de los clientes.

5.1.1 Descarga e instalación de Hadoop

Descargaremos e instalaremos Hadoop (*versión 2.7.2*) en una instancia EC2 t2.micro de AWS. En este caso, será una imagen de Centos 7 (*la instalación en Ubuntu puede dar problemas*). Esta imagen podemos buscarla en el AWS marketplace (gratuita). Los únicos requisitos que tiene Hadoop son: tener instalado Java y SSH. Descargaremos Java de la misma forma que lo hicimos para Apache Spark (ahora con el comando `yum` en lugar de `apt-get`), modificando el fichero `/etc/profile` con el `JAVA_HOME` (y el posterior `source /etc/profile`), así como las líneas:

```
PATH=$PATH:$JAVA_HOME/bin

export JAVA_HOME
```

También descargaremos `vim`, para poder editar los ficheros de una forma cómoda. Ya podemos descargar Hadoop (podemos encontrar el enlace de descarga en http://apache.rediris.es/hadoop/common/hadoop-2.7.2/):

```
curl -O dir_descarga_hadoop

tar -xzvf hadoop-2.7.2.tar.gz
```

Ahora, nos movemos al directorio que aparece al descomprimir el fichero que acabamos de descargar, y tendremos que modificar un fichero de configuración, para añadir el `JAVA_HOME` de nuevo. Se trata del fichero **`/etc/hadoop-env.sh`**, y debemos añadir:

```
export JAVA_HOME=/usr/lib/jvm/jre-1.7.0-openjdk
```

Podemos probar a ejecutar el siguiente comando, que mostrará la documentación de uso de Hadoop si todo ha ido bien:

```
bin/hadoop
```

5.1.2 Configuración de SSH

Hadoop utiliza SSH para correr los scripts que gestionan los demonios remotos de Hadoop. Deberíamos ser capaces de conectarnos mediante `ssh localhost` sin contraseña. En caso de que no sea así, debemos ejecutar los siguientes comandos:

```
ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa

cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys

chmod 0600 ~/.ssh/authorized_keys
```

Con esto, ya deberíamos ser capaces de conectarnos sin contraseña.

5.1.3 Configuración de HDFS

Debemos especificar la dirección en la que correrá HDFS, ya que por defecto estará en localhost, y queremos que sea accesible desde cualquier lugar. Para ello, tendremos que modificar dos ficheros.

El primero de estos ficheros es **`core-site.xml`** en `/etc/hadoop`. Añadiremos el siguiente trozo de código entre las etiquetas de `<configuration>`:

```
<property>

     <name>fs.default.name</name>
     <value>hdfs://dirección_IP:9000</value>

</property>
```

El otro fichero que debemos modificar es **`slaves`**, en `/etc/hadoop`. Aquí especificaremos que el DataNode está corriendo en la propia instancia. Es en este fichero en el que se debe indicar todos los esclavos (DataNodes) que tendrá el clúster HDFS. En nuestro caso sólo habrá un DataNode corriendo en la propia máquina. En el fichero slaves, debemos añadir la dirección IP o el hostname de los DataNodes.

5.1.4 Ejecución de HDFS

Tendremos que seguir los siguientes pasos:

1) Formatear el sistema de ficheros:

```
bin/hdfs namenode -format
```

2) Arrancar los demonios del NameNode y del DataNode:

```
sbin/start-dfs.sh
```

3) Podemos acceder al UI web del NameNode (por defecto, puerto 50070):

```
http://IP:50070
```

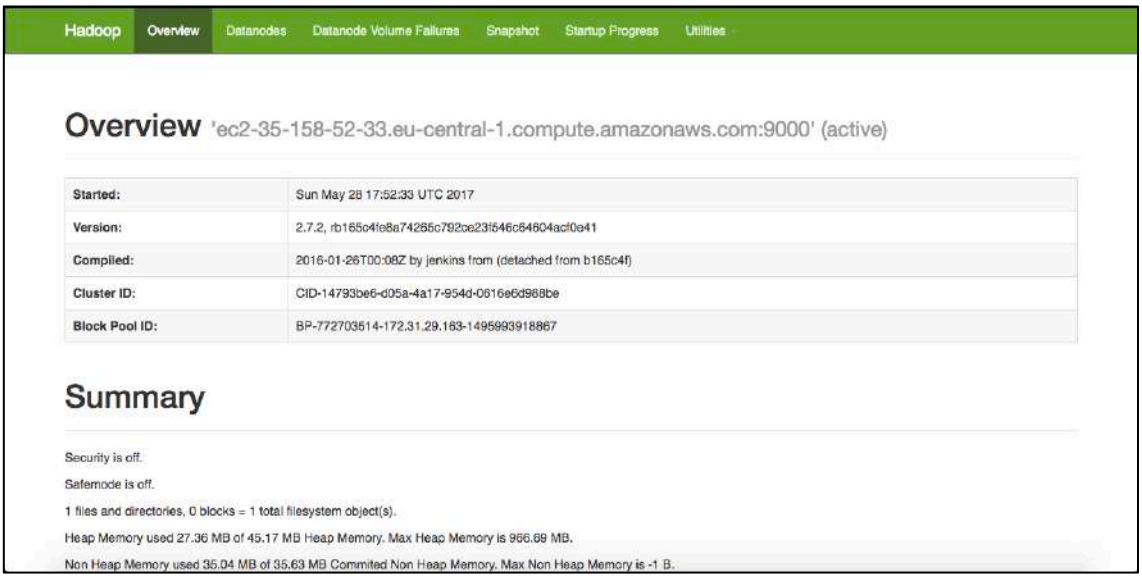

Figura 36. Interfaz web HDFS

4) Crear los directorios de HDFS requeridos para ejecutar las tareas de MapReduce (en nuestro caso, *username=centos*):

```
bin/hdfs dfs -mkdir /user

bin/hdfs dfs -mkdir /user/<username>
```

5) Copiar ficheros de datos al sistema de ficheros distribuido:

```
bin/hdfs dfs -put directorio/datos /user/<username>
```

6) Si queremos parar los demonios, ejecutamos:

```
sbin/stop-dfs.sh
```

Una vez que hayamos añadido los ficheros de datos al sistema de ficheros, podremos verlo en la interfaz web, en la pestaña *Utilities* → *Browse the file system*.

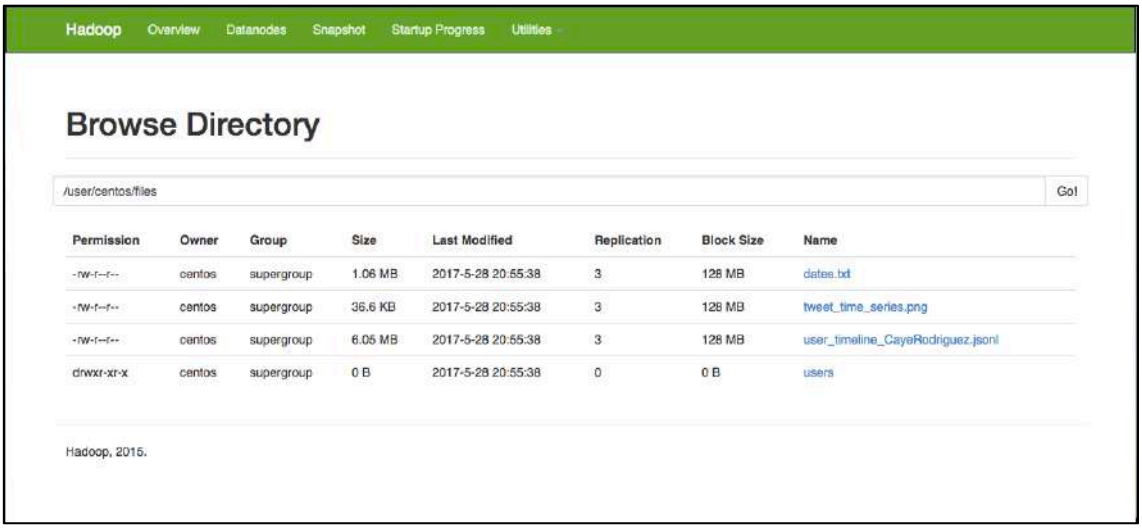

Figura 37. Ficheros almacenados en HDFS

Con esto, ya tenemos nuestro clúster HDFS funcionando y accesible a través de la red. Durante el desarrollo de este proyecto, se accederá programáticamente a este sistema de ficheros, tanto para almacenar ficheros como para recuperarlos.

5.2 Automatización

La tarea de añadir más workers a plataforma de procesamiento de datos es bastante tediosa, teniendo que crear una nueva instancia EC2 de Amazon Web Services, instalar todos los prerrequisitos y finalmente instalar Apache Spark y arrancar un worker que se conectará al nodo máster. Podemos simplificar en gran medida esta tarea mediante la automatización. Vamos a utilizar dos de las herramientas más relevantes en este ámbito: Docker y Ansible.

5.2.1 Docker

Docker automatiza las tareas repetitivas de crear y configurar entornos de desarrollo, de forma que los desarrolladores puedan centrarse en las aplicaciones.

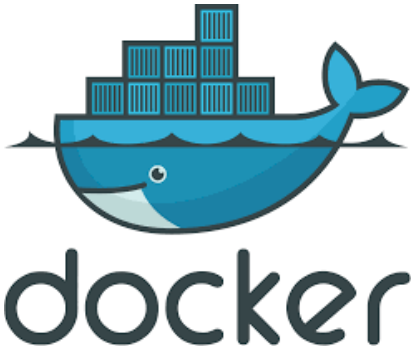

Figura 38. Logo Docker

Docker se basa en el concepto de ***contenedores***. Con el uso de estos contenedores, todo lo necesario para hacer que un trozo de software funcione, se encapsula en contenedores aislados. A diferencia de las máquinas virtuales, los contenedores no contienen un sistema operativo completo: sólo necesitan algunas librerías y configuraciones para que el software funcione. Esto permite asegurar que el software siempre se ejecutará del mismo modo, independientemente de dónde se despliegue.

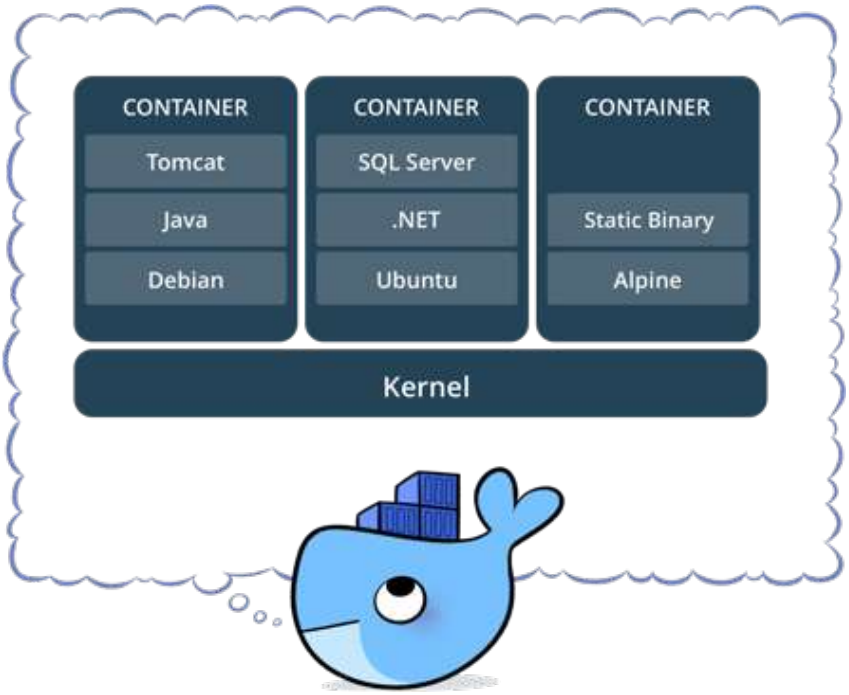

Figura 39. Contenedor Docker

Docker permite construir imágenes automáticamente, leyendo instrucciones de un ***Dockerfile***, un fichero de texto que contiene todos los comandos, en orden, necesarios para construir una imagen dada. Los Dockerfiles tienen un formato y estructura muy concretos.

Para realizar una automatización del escenario, se creará un Dockerfile que permitirá la creación de contenedores con la imagen especificada en el Dockerfile, que descargará los paquetes necesarios, instalará Apache Spark y arrancará un worker, conectado al máster automáticamente. Así, será posible crear y arrancar nuevos workers únicamente ejecutando un nuevo contenedor con la imagen previamente descrita.

Instalación de Docker

Podemos descargar Docker en la instancia muy fácilmente con el comando:

```
sudo apt-get install docker.io
```

No es necesario realizar ninguna configuración adicional, con esto bastaría para poder ejecutar Docker.

Dockerfile

El siguiente paso para crear nuestra imagen será escribir un Dockerfile con las especificaciones que deseemos. En nuestro caso, el objetivo será instalar y arrancar Apache Spark, por lo que habrá que descargar los paquetes que necesite Spark para funcionar.

En primer lugar, creamos un nuevo directorio, donde ubicaremos nuestro Dockerfile, y creamos un fichero vacío:

```
mkdir spark-worker

cd spark-worker

touch Dockerfile
```

A continuación se muestra el contenido del Dockerfile. Podemos acceder a él desde el enlace https://hub.docker.com/r/alerguez/spark-worker/~/dockerfile/.

```
FROM ubuntu:16.04
MAINTAINER alejandro.rodriguez.calzado@alumnos.upm.es cayetano.rodriguez.medina@alumnos.upm.es

RUN apt-get update \
  && apt-get install -y openjdk-8-jdk \
  && apt-get install -y python2.7 \
  && apt-get install -y python-minimal \
  && apt-get install -y scala

# JAVA
ENV JAVA_HOME /usr/lib/jvm/java-1.8.0-openjdk-amd64

WORKDIR /opt

# Download Spark 2.1.0 pre-build for Hadoop 2.1 and later
ADD http://d3kbcqa49mib13.cloudfront.net/spark-2.1.0-bin-hadoop2.7.tgz ./

RUN tar -xvzf spark-2.1.0-bin-hadoop2.7.tgz \
  && rm spark-2.1.0-bin-hadoop2.7.tgz


WORKDIR /opt/spark-2.1.0-bin-hadoop2.7

CMD ./bin/spark-class org.apache.spark.deploy.worker.Worker spark://sparkmaster.ddns.net:7077
```

Figura 40. Dockerfile

La primera línea especifica la imagen que debe usarse como base, para ello se usa la palabra clave **`FROM`**. La siguiente línea especifica los autores del Dockerfile (palabra clave **`MAINTAINER`**). Esta línea no es ejecutada por Docker, simplemente es informativa.

A continuación, con la palabra clave **`RUN`** se describen los comandos que deben ejecutarse en la imagen base. Básicamente se trata de instalar los paquetes requeridos por Spark para funcionar (Java, Python y Scala). Como ya sabemos, es necesario añadir el `JAVA_HOME`. En el Dockerfile, esto se hace con la palabra clave **`ENV`**.

Usaremos la palabra clave **`WORKDIR`** para cambiarnos de directorio (similar al comando `cd`). Lo hacemos para descargar Spark en el directorio `/opt`. La descarga puede hacerse con la palabra clave **`ADD`**. El `ADD` acepta dos parámetros (origen y destino), y funciona de la siguiente manera: copia un fichero origen (en el host, no el contenedor) en un directorio destino del contenedor. Si se especifica una URL en lugar de un directorio origen, lo que hará será descargar el contenido de la URL en el directorio destino.

Posteriormente, volvemos a usar `RUN` para descomprimir el fichero que acabamos de descargar y para eliminar este fichero.

Para finalizar, volvemos a cambiar de directorio para posicionarnos en el recién creado, y usamos la palabra clave **`CMD`** para especificar los comandos que queramos ejecutar en el contenedor una vez esté creado. En nuestro caso, querremos arrancar un worker y enlazarlo al máster.

Docker Image

Una vez tenemos nuestro Dockerfile, el siguiente paso será crear una imagen a partir de él. Esto podemos hacerlo con el comando:

```
docker build -t alerguez/spark-worker:latest .
```

Este comando busca un fichero llamado Dockerfile en el directorio actual. Se usa la etiqueta `latest` para utilizar la última versión del Dockerfile. Podemos facilitar la tarea de creación de la imagen descargándonos la imagen base de Docker Hub, de modo que la encuentre localmente y no tenga que buscarla.

```
docker pull ubuntu:16.04
```

El `docker build` lee el Dockerfile y procesa, una a una, las instrucciones para crear una imagen llamada `alerguez/spark-worker`.

Con esto, ya tenemos una imagen que podemos ejecutar en un contenedor. Para comprobar que se ha creado correctamente, podemos consultar las imágenes existentes en nuestro ordenador con el comando:

```
docker images
```

Docker Hub: subir y bajar imágenes

Ahora, podemos subir la imagen que hemos creado a Docker Hub. En primer lugar, tendremos que hacer login en Docker Hub desde línea de comandos:

```
docker login
```

Se nos pedirá un usuario y contraseña. Finalmente, podemos subir la imagen con el comando:

```
docker push alerguez/spark-worker
```

Igualmente, si queremos descargarnos la imagen, podemos hacerlo de la siguiente forma:

```
docker pull alerguez/spark-worker
```

Esto no es estrictamente necesario, ya que si no tenemos la imagen en nuestra máquina local, al intentar ejecutar un contenedor con la imagen, Docker la descargará automáticamente de Docker Hub.

Docker Hub: Automate Build

En el subapartado anterior hemos explicado el proceso de creación de una imagen de forma manual. Utilizando Docker Hub, es posible realizar este mismo proceso de forma automática. Para hacer esto, necesitaremos enlazar la cuenta de Docker Hub con GitHub. Esto podemos hacerlo en `Profile > Settings > Linked Accounts & Services`.

El "*Automate Build*" se basa en la integración con el código existente en el repositorio de GitHub. Nuestro código está en el repositorio https://github.com/alerodcal/spark-worker. En Docker Hub, pulsamos sobre `Create > Create Automate Build`. Ahí tendremos que elegir el proyecto de GitHub que queremos automatizar.

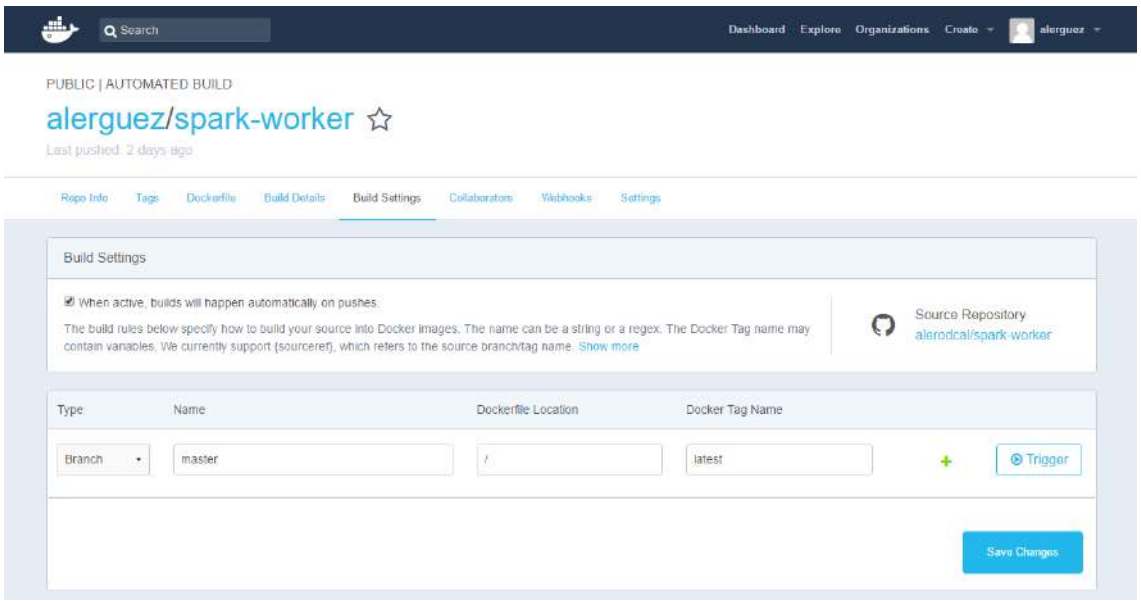

Figura 41. Docker Hub

En la imagen anterior, podemos ver la configuración realizada sobre esta imagen automatizada. Hemos activado la opción de actualizar automáticamente tras cada `push` en el repositorio de GitHub, para que los cambios se actualicen de manera

inmediata y automática. También hemos especificado que la rama sobre la que se debe hacer la automatización (rama `master`), y la etiqueta a usar (por defecto, `latest`).

La primera vez que creamos un "*Automated Build*", Docker Hub crea la imagen. Tras unos minutos podremos ver la imagen en el dashboard.

Ejecución de un contenedor

Ya disponemos de la imagen que hemos diseñado para crear nuevos workers de Spark. Lo único que nos queda es ejecutar la imagen en un contenedor de Docker.

```
docker run -d -P --name worker --net=host alerguez/spark-worker:latest
```

Usamos la opción **`-d`** para ejecutar el contenedor en background, de modo que no muestre la salida en primer plano.

La opción **`-P`** publica todos los puertos del host, realizando un enlace entre los puertos del contenedor y el host.

Especificamos el nombre del contenedor con la opción **`--name`**.

Y, por último, la opción **`--net=host`** es muy importante, ya que permite conectividad completa entre el contenedor y el host, de modo que se permite la comunicación entre el contenedor y el resto de Internet, atravesando el host sobre el que corre el contenedor.

5.2.2 Ansible

Una vez tenemos la imagen de Docker preparada para correr un worker de Spark en cualquier instancia, el siguiente paso sería conectarnos a cada instancia, instalar Docker, descargar la imagen del repositorio DockerHub, crear y ejecutar un contenedor a partir de esta imagen. Esto puede ser una tarea pesada si tenemos que desplegar muchos workers, por lo que decidimos añadir un nivel más de automatización mediante una herramienta como Ansible.

Ansible es una herramienta *open source* desarrollada en Python y que podemos definir como un motor de orquestación para la automatización de tareas. Ansible nos permitirá desplegar, de forma simultánea en cualquier número de instancias, workers de Spark a partir de la imagen de Docker que hemos creado. Las dos grandes ventajas que hemos encontrado en esta herramienta frente a otras de la misma índole, como Chef, es que se conecta a los clientes por SSH de forma paralela, por lo que la conexión es muy sencilla y que no es necesario instalar ningún tipo de agente en los clientes.

Figura 42. Logo Ansible

Instalación de Ansible

Como hemos dicho, Ansible no necesita instalar ningún agente en los nodos que van ser gestionados, siendo el único requisito que los nodos cuenten con una versión de Python superior a la 2.5. En caso de que esta versión no venga por defecto en las máquinas gestionadas, no es necesario conectarse a cada una de ellas para instalarlo ya que Ansible cuenta con el módulo **`raw`** que nos permite ejecutar comandos de la shell en los nodos gestionados. Este módulo no necesita de Python para realizar su cometido, por lo que puede ser usado para instalarlo y, de hecho, lo hemos usado para ello.

La otra parte de la arquitectura de Ansible la compone la máquina de gestión, que es la encarda de enviar las órdenes a los nodos gestionados. El único requisito para la instalación de Ansible en ésta es Python versión 2.6 o 2.7. En nuestro caso hemos usado un ordenador con Ubuntu 16.04. A continuación, pasamos a detallar los comandos ejecutados para la instalación de Ubuntu en este sistema:

```
sudo apt-get install software-properties-common

sudo apt-add-repository ppa:ansible/ansible

sudo apt-get update

sudo apt-get install ansible
```

Conexión con los nodos

Para que Ansible sea capaz de conectarse a los nodos y enviar órdenes, el único requisito es tener conectividad mediante SSH con ellos. Para ello es necesario realizar algunos pasos para que nuestro sistema sepa que clave privada debe usar para conectarse a cada nodo sin necesidad de especificar una clave de acceso.

En nuestro caso hemos usado la misma clave privada para todas las instancias de Amazon Web Services. El primer paso es copiar la clave en formato "`pem`" en el directorio **`~/.ssh/`**.

```
cp key-spark.pem ~/.ssh/
```

A continuación, debemos modificar el fichero de configuración de SSH para añadir los parámetros de cada nodo. Por cada nodo debemos añadir a este fichero las siguientes líneas:

```
Host <IDENTIFICADOR-NODO>

   Hostname <DNS-PUBLICO>

   User <USUARIO>

   IdentityFile <RUTA-AL-FICHERO-PEM/NOMBRE-PEM>
```

Una vez hecho esto debemos añadir al fichero **`/etc/ansible/hosts`** el identificador que usamos en el archivo anterior para cada uno de los nodos. Los nodos se engloban en un grupo para poder referirnos a todos ellos simultáneamente como objetivos de nuestro ***playbook***. El formato que debemos usar en este fichero es el siguiente:

```
[<NOMBRE-GRUPO>]

<IDENTIFICADOR-NODO-1>

<IDENTIFICADOR-NODO-2>

...

<IDENTIFICADOR-NODO-N>
```

Debido a que los nombres DNS que asigna Amazon Web Services automáticamente a sus instancias son demasiado largos, debemos añadir al fichero de configuración de Ansible (**`/etc/ansible/ansible.cfg`**) la siguiente línea tras la sentencia `[ssh_connection]`:

```
Control_path = %(directory)s/%%h-%%r
```

Con esto, Ansible ya es capaz de enviar las órdenes de nuestro playbook a todos los nodos gestionados. Para ejecutar un playbook basta con usar el siguiente comando en el directorio donde esté almacenado dicho playbook:

```
ansible-playbook worker-playbook.yml
```

Playbook

Para la automatización del despliegue mediante Ansible, hemos escrito un playbook que se encarga de instalar en los nodos todas las dependencias necesarias para ejecutar un contenedor de Docker basado en la imagen que hemos creado. El contenido de este playbook se muestra a continuación:

```
---

- hosts: spark-workers

  remote_user: ubuntu

  gather_facts: False

  environment:

      PYTHONPATH: "{{ lookup('env','PYTHONPATH')
}}:/home/ubuntu/.local/lib/python2.7/site-
packages:/usr/local/lib/python2.7/dist-
packages:/usr/local/lib/python2.7/site-packages”

  tasks:
    - name: Install python-minimal

      raw: test -e /usr/bin/python || (apt-get -y update &&
apt-get install -y python-minimal)

      become: yes

      become_method: sudo

    - name: Install (or update) docker.io

      apt:

        name: docker.io
```

```
      state: latest
      update_cache: yes
      become: yes
      become_method: sudo
  - name: Install python-pip
    apt:
      name: python-pip
      state: latest
      update_cache: yes
      become: yes
      become_method: sudo
  - name: Install certain python modules for Docker
    pip:
      name: "{{ item.name }}"
      version: "{{ item.version }}"
      state: present
      with_items:
        - { name: docker, version: 2.0.0 }
        - { name: docker-py, version: 1.10.6 }
  - name: Pull alerguez/spark-worker image
    docker_image:
      name: alerguez/spark-worker
      become: yes
      become_method: sudo
```

```
-  name: Create a data container

        docker_container:

          name: spark-worker

          image: alerguez/spark-worker

          detach: yes

          network_mode: host

          published_ports:

            - all

          become: yes

          become_method: sudo

...
```

Figura 43. Ansible Playbook

El playbook comienza indicando el grupo de los definidos en `/etc/ansible/hosts` que van a ejecutar este playbook, mediante la palabra **`hosts`**. Seguidamente mediante **`remote_user`** podemos indicar el usuario que ejecutará estas acciones en los hosts y usando **`environment`** podemos definir variable de entorno que estarán disponibles durante la ejecución del playbook. En este caso hemos definido la ruta a la instalación de Python y sus módulos.

Una vez definidos los parámetros generales, con **`tasks`**, comienza una lista de tareas que componen lo que realmente se ejecutará en los nodos con este playbook. Cada tarea se le asigna un nombre mediante **`name`** y hace uso de un módulo de Ansible según la tarea. Las sentencias **`become`** y **`become_method`** sirven para ejecutar la tarea como usuario con privilegios.

La primera tarea consiste en instalar Python que, como dijimos, es el único requisito para los nodos gestionados mediante Ansible. La instalación se ejecuta mediante el módulo **`raw`** que nos permite ejecutar comandos de la Shell sin necesidad de tener Python instalado. La tarea comprueba si Python está instalado y en caso de que no lo esté realiza un `apt-get update` y lo instala mediante `apt-get install`.

La siguiente tarea se encarga de instalar la última versión de Docker en los nodos gestionados para ello hace uso del módulo **`apt`** de Ansible. Previamente a la instalación se realiza un `apt update`.

El módulo que vamos a usar para ejecutar el contenedor de Docker necesita de algunos módulos de Python en los nodos gestionados. Para la instalación de estos módulos se usa la herramienta **`pip`.** Esta herramienta se instala mediante el módulo **`apt`** en la siguiente tarea.

A continuación, en la tarea con nombre "*Install certain python modules for docker*" vamos a instalar los dos módulos de Python necesarios: Docker y Docker-py. Para ello se hace uso del módulo **`pip`**. El nombre y la versión de módulos se indica mediante el uso de ítems. Con la directiva **`with_items`** se indican con que ítems deben ejecutarse la tarea.

Con esto ya tenemos Docker en nuestros nodos y somos capaces de gestionarlos mediante los módulos de Ansible diseñados para ello. En la tarea "*Pull alerguez/spark-worker image*" se hace uso del módulo **`docker_image`** para descargar de Docker Hub en todos los nodos la imagen que hemos creado.

Una vez tenemos disponible la imagen en los nodos, el último paso es ejecutar un contenedor Docker a partir de esta imagen. Esto se realiza en la última tarea mediante el módulo **`docker_container`**.

Este playbook se ha probado con hasta 10 nodos simultáneamente, consiguiendo que en cuestión de poco más de 2 minutos, los 10 nodos se uniesen correctamente al cluster de Spark, listos para ejecutar las tareas que el master ordene. Gracias a que las tareas se realizan en paralelo en todos los nodos, Ansible tarda lo mismo si ejecutamos el playbook en un nodo que en varios, lo que demuestra la gran ventaja que supone la gestión de nodos mediante esta herramienta.

5.3 DNS dinámico

Cuando lanzamos una instancia EC2 de Amazon Web Services, se le asigna una dirección IP pública y un nombre de DNS público. Podremos usar ambos para acceder a la instancia. Sin embargo, y debido a la cantidad de hosts existentes en el dominio de AWS, estos nombres tienen que tener una cierta longitud para poder ser únicos. Un nombre de DNS público para una instancia EC2 tiene la siguiente forma: **`ec2-12-34-56-78.us-west-2.compute.amazonaws.com`**, donde se incluye:

- Dominio de Amazon Web Services
- El servicio concreto, en este caso compute

- La región
- La dirección IP pública de la instancia

Existen servicios de DNS dinámicos, que proveen de hostnames personalizados, con su dominio correspondiente, y que son mucho más fáciles de recordar y de utilizar para alcanzar nuestro servidor. Además, algunos de estos servicios son gratuitos. Podemos utilizar un proveedor de DNS dinámico junto con nuestra instancia EC2, y configurar la instancia para actualizar las direcciones asociadas con un nombre de DNS público cada vez que la instancia se arranque. Estudiaremos el caso de utilizar el proveedor No-IP, uno de los servicios gratuitos existentes en el mercado, para registrar un hostname para nuestros servidores en los que estarán corriendo el nodo máster de Spark y el clúster HDFS, y configurar un servidor DNS dinámico en ambos, así como para otras posibles máquinas que se incluyan en el escenario.

5.3.1 Registrar un hostname

Para poder utilizar los servicios de No-IP, tenemos que crear una cuenta en su página web https://www.noip.com/sign-up. Aquí, podemos ya especificar el hostname que nos gustaría registrar, junto con el dominio concreto de entre una lista de dominios gratuitos. Añadiremos también un correo electrónico, un nombre de usuario y una contraseña. Una vez completada la creación de la cuenta en No-IP, tendremos que confirmarla mediante un correo electrónico, y ya habremos registrado con éxito nuestro hostname. Los hostnames registrados en nuestro caso son:

- Máster Spark: `tfmspark.ddns.net`
- Clúster HDFS: `tfmhdfs.ddns.net`

5.3.2 Instalación del Dynamic Update Client (DUC)

A continuación, tenemos que instalar el DUC de No-IP para Linux. Podremos hacerlo de una forma muy sencilla mediante el terminal, tras conectarnos a la instancia correspondiente. Todos los comandos que tendremos que introducir, necesitarán permisos de superusuario, por lo que podemos hacer login como usuario root, o bien, utilizar sudo con todos los comandos.

En primer lugar, tenemos que descargarnos el código del DUC.

```
cd /usr/local/src/

wget http://www.no-ip.com/client/linux/noip-duc-linux.tar.gz
```

Ahora, extraemos el fichero comprimido que acabamos de descargar y comenzamos la instalación.

```
tar xf noip-duc-linux.tar.gz

cd noip-2.1.9-1/

make install
```

Es posible que sea necesario instalar `make` y `gcc` para poder realizar la instalación.

```
apt-get install make

apt-get install gcc
```

Por último, tendremos que completar la información que se nos solicite por pantalla. Como podemos ver en la siguiente imagen, se pedirá el login/email y la contraseña. A continuación, en caso de que exista más de un hostname registrado en nuestra cuenta tendremos que elegir cuál de ellos usar (en nuestro caso, sólo tenemos uno, por lo que se utilizará ese de forma automática), junto con un intervalo de actualización del hostname (que será el plazo en días, tras el que se nos preguntará si queremos seguir conservando el hostname o no), y si queremos ejecutar algo tras estas actualizaciones.

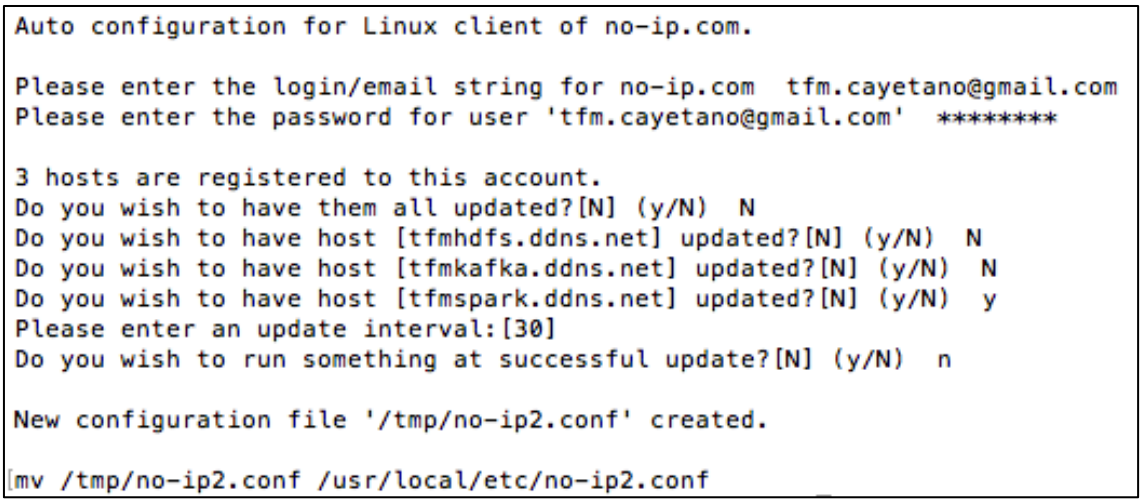

Figura 44. Configuración de No-IP

Podremos configurar nuevamente el DUC, en caso de que fuera necesario en algún momento, en el fichero **`/usr/local/etc/no-ip2.conf`**, que es el fichero de configuración por defecto. Es posible utilizar otra ruta para este fichero, y se especificaría al arrancar No-IP con la opción `-c`.

5.3.3 Configuración de puertos

Antes de poder arrancar el servicio de No-IP, tendremos que abrir el puerto que utiliza para su tráfico. Se trata del puerto 8245 (TCP), y será necesario abrirlo para tráfico entrante y saliente. Para ello, haremos uso de los grupos de seguridad en el dashboard de AWS.

Añadiremos una regla TCP personalizada, para el puerto 8245, y que acepte tráfico entrante desde cualquier dirección (el tráfico saliente está configurado para permitirlo todo, independientemente del protocolo o puerto utilizado). Esto es lo que podemos ver en la imagen siguiente:

Figura 45. Grupo de seguridad para el puerto de No-IP

5.3.4 Arranque de No-IP

Para arrancar No-IP, simplemente tenemos que ejecutar el siguiente comando:

```
/usr/local/bin/noip2
```

Con este comando, se buscará el fichero de configuración en la ruta por defecto `/usr/local/etc/no-ip2.conf`. Como ya se ha mencionado previamente, podemos utilizar otro fichero de configuración de la siguiente forma:

```
/usr/local/bin/noip2 -c /ruta/a/fichero_de_conf/no-ip2.conf
```

Una vez arrancado, ya podremos acceder a nuestro servidor utilizando los hostnames que registramos anteriormente. Así, podremos arrancar los workers de Spark especificando la dirección del máster de la siguiente forma:

```
spark://tfmspark.ddns.net:7077
```

Del mismo modo, para acceder al sistema de ficheros de HDFS, sólo necesitaremos añadir la dirección de la siguiente forma:

```
hdfs://tfmhdfs.ddns.net:9000
```

5.4 Script de arranque automático de HDFS

A la hora de arrancar el sistema de almacenamiento distribuido HDFS, era necesario modificar el fichero `core-site.xml` con la dirección del DNS público de Amazon Web Services de la instancia en la que estaba corriendo. Cada vez que se arranca la instancia, este DNS cambia, con lo cual era una tarea tediosa tener que modificarlo cada vez que se paraba y arrancaba la instancia. Así, se decidió realizar un script que modificara automáticamente el fichero mencionado con la nueva dirección, y arrancara de paso el clúster.

```
#!/bin/bash

case "$1" in
    hadoop-start)
    cp /home/centos/hadoop-2.7.2/etc/hadoop/core-site.xml /home/centos/hadoop-2.7.2/etc/hadoop/core-site.xml.bck
    HDFS_IP=`nslookup sparkhdfs.ddns.net | grep Address | tail -n 1 | grep -Eo [0-9,.]* | sed 's/\./-/g'`
    cat /home/centos/hadoop-2.7.2/etc/hadoop/core-site.xml | sed "s/ec2-[0-9,-]*/ec2-${HDFS_IP}/g"
        > /home/centos/hadoop-2.7.2/etc/hadoop/core-site.xml

    /home/centos/hadoop-2.7.2/sbin/start-dfs.sh
    ;;
    dns)
     sudo /usr/local/bin/noip2
    ;;
    hadoop-stop)
     /home/centos/hadoop-2.7.2/sbin/stop-dfs.sh
    ;;
    *)
    echo "Usage: $0 {hadoop-start|hadoop-stop|dns}"
    exit 1
    ;;
esac

exit 0
```

Figura 46. Script de arranque automático de HDFS

En primer lugar, se realiza una copia de seguridad del fichero `core-site.xml`, ya que si no está actualizado el DNS dinámico, se borraría el fichero por completo. A continuación se extrae la nueva dirección IP, realizando una consulta al DNS mediante el comando `nslookup`. A la respuesta de este comando, habrá que realizarle una serie de modificaciones, para quedarnos únicamente con la dirección IP que nos interesa. Una vez tenemos la nueva dirección IP, modificamos el fichero `core-site.xml`, en el que sólo habrá que cambiar la línea en la que aparece la antigua dirección y poner la nueva. Hay que tener en cuenta el formato en el que aparece esta dirección, y es que hay que usar el formato de las direcciones de DNS público de AWS. En este formato, la dirección IP pública aparece tras la secuencia "*ec2-*", con lo cual, insertaremos la nueva dirección en esa posición. Cuando el fichero esté modificado, se arrancará HDFS con el script `sbin/start-dfs.sh`.

Se ha aprovechado este script para incluir una opción que arranque el DNS dinámico, y otra opción para parar HDFS.

Es posible que comando `nslookup` no venga instalado, dependiendo de la versión de sistema operativo que estemos usando, en CentOS puede instalarse mediante:

```
sudo yum install bind-utils
```

5.5 Escenario completo

Al añadir todas estas mejoras al escenario básico, obtenemos una solución más completa y escalable, pudiendo añadir cualquier número de workers que se conectarán automáticamente al nodo máster de Spark y se unirán a la ejecución de las aplicaciones de manera instantánea. El uso de un servicio de DNS dinámico facilita mucho la conexión con las máquinas más importantes de nuestro escenario: el nodo máster de Spark y el sistema de almacenamiento distribuido HDFS. También podría ser usado en un futuro para otras máquinas de especial interés.

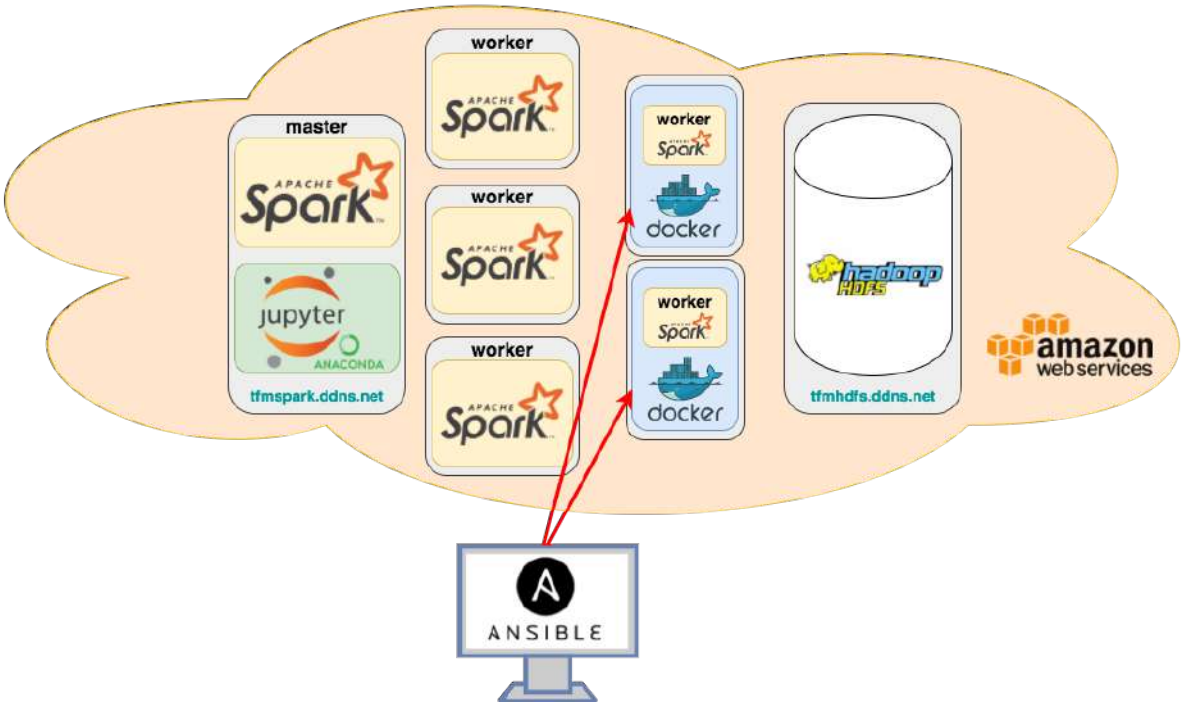

Figura 47. Escenario completo de plataforma de procesamiento de datos

6 Procesamiento de datos y análisis de información

En este capítulo abordaremos la última etapa en el proceso de Big Data: el procesamiento de los datos obtenidos de las fuentes de datos y el análisis de la información generada a partir de este procesado de los datos. Para ello, y con el fin de investigar y ahondar más en los módulos de Apache Spark, se ha diseñado una serie de aplicaciones, cada una de las cuales extraerá información de una fuente de datos, realizará un procesamiento utilizando las tecnologías de Spark, y generará ciertos resultados y gráficas que podrán analizarse fácilmente. Se ha considerado de especial interés explotar al máximo el potencial de Apache Spark, por lo cual, cada una de las aplicaciones desarrolladas se ha implementado utilizando diferentes módulos de Spark.

6.1 Análisis completo de usuario

La primera aplicación desarrollada se centra en realizar un análisis completo de un usuario de la red social. Este análisis comprende distintas fases: análisis de personas, análisis de influencia del usuario, y análisis de términos más usados por el mismo.

Para realizar estos análisis, necesitaremos obtener los datos de la API de Twitter, y para ello se han diseñado dos aplicaciones Python con el fin de almacenar, en ficheros con formato JSON, la lista de seguidores y amigos del usuario en cuestión, así como el conjunto de todos los tweets publicados por el mismo.

Obtendremos la lista de seguidores y amigos de un usuario mediante la ejecución de la aplicación Python `twitter_get_user.py`. Para mostrar el contenido de la aplicación, se ha dividido en tres partes: una parte común, una parte para la obtención de los seguidores, y otra parte para la obtención de los amigos.

En este primera parte común, se realiza la autenticación con la API de Twitter, utilizando las credenciales OAuth. Una vez hecho esto, se intenta crear el directorio `users`, para incluir en su interior un directorio por cada usuario al que se quiera realizar este análisis. En este directorio habrá, por tanto, un directorio cuyo nombre coincidirá con el nombre del usuario que se quiera analizar. Finalmente, dentro de cada uno de estos directorios individuales, se localizarán los ficheros en formato JSON con la lista de seguidores y amigos del usuario en cuestión, `followers.jsonl` y `friends.jsonl` respectivamente.

twitter_get_user.py

```python
import os
import sys
import json
import time
import math
from tweepy import Cursor, API, OAuthHandler

MAX_FRIENDS = 15000

def usage():
  print("Usage:")
  print("python {} <username>".format(sys.argv[0]))

def paginate(items, n):
  """Generate n-sized chunks from items"""
  for i in range(0, len(items), n):
    yield items[i:i+n]

if __name__ == '__main__':
  if len(sys.argv) != 2:
    usage()
    sys.exit(1)
  screen_name = sys.argv[1]

  ACCESS_TOKEN = 'XXXXXXXXXXXXXXXX'
  ACCESS_SECRET = 'XXXXXXXXXXXXXXXX'
  CONSUMER_KEY = 'XXXXXXXXXXXXXXXX'
  CONSUMER_SECRET = 'XXXXXXXXXXXXXXXX'

  auth = OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)
  auth.set_access_token(ACCESS_TOKEN, ACCESS_SECRET)
  client = API(auth)
  dirname = "users/{}".format(screen_name)
  max_pages = math.ceil(MAX_FRIENDS / 5000)
  try:
    os.makedirs(dirname, mode=0o755, exist_ok=True)
  except OSError:
    print("Directory {} already exists".format(dirname))
  except Exception as e:
    print("Error while creating directory {}".format(dirname))
    print(e)
    sys.exit(1)
```

Figura 48. Fichero twitter_get_user.py (1)

Lo siguiente que se lleva a cabo es realizar una serie de peticiones a la API de Twitter en relación a la lista de seguidores del usuario. Si hay más de 5000 seguidores, se espera un minuto entre peticiones para evitar el rate limit.

```python
# get followers for a given user
fname = "users/{}/followers.jsonl".format(screen_name)
with open(fname, 'w') as f:
  for followers in Cursor(client.followers_ids,
                          screen_name=screen_name).pages(max_pages):
    for chunk in paginate(followers, 100):
      users = client.lookup_users(user_ids=chunk)
      for user in users:
        f.write(json.dumps(user._json)+"\n")
    if len(followers) == 5000:
      print("More results available. Sleeping for 60 seconds to  avoid rate limit")
      time.sleep(60)
```

Figura 49. Fichero twitter_get_user.py (2)

Se crea el fichero `followers.jsonl` que contiene un objeto de tipo *User* por cada seguidor que tenga el usuario.

Por otro lado, se hace lo mismo para los amigos del usuario (personas a las que sigue), como podemos ver en la siguiente imagen. En este caso se crea el fichero `friends.jsonl` con los amigos del usuario, representados por objetos *User*.

```
57    # get friends for a given user
58    fname = "users/{}/friends.jsonl".format(screen_name)
59    with open(fname, 'w') as f:
60      for friends in Cursor(client.friends_ids,
61                            screen_name=screen_name).pages(max_pages):
62        for chunk in paginate(friends, 100):
63          users = client.lookup_users(user_ids=chunk)
64          for user in users:
65            f.write(json.dumps(user._json)+"\n")
66        if len(friends) == 5000:
67          print("More results available. Sleeping for 60 seconds to avoid rate limit")
68          time.sleep(60)
```

Figura 50. Fichero twitter_get_user.py (3)

Al ejecutar esta aplicación, se tendremos los ficheros `followers.jsonl` y `friends.jsonl` en el directorio `/users/nombre_usuario`. Se accederá a estos ficheros desde la aplicación Spark en Jupyter para convertirlos en RDDs y poder trabajar con ellos cómodamente.

Del mismo modo, también se ha diseñado una aplicación Python, **`twitter_get_user_timeline.py`**, que almacena en un fichero JSON todos los tweets publicados por el usuario.

twitter_get_user_timeline.py

```
1   import sys
2   import json
3   from tweepy import Cursor, API, OAuthHandler
4
5   if __name__ == '__main__':
6     user = sys.argv[1]
7
8     ACCESS_TOKEN = 'XXXXXXXXXXXXXXXX'
9     ACCESS_SECRET = 'XXXXXXXXXXXXXXXX'
10    CONSUMER_KEY = 'XXXXXXXXXXXXXXXX'
11    CONSUMER_SECRET = 'XXXXXXXXXXXXXXXX'
12
13    auth = OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)
14    auth.set_access_token(ACCESS_TOKEN, ACCESS_SECRET)
15    client = API(auth)
16
17    fname = "user_timeline_{}.jsonl".format(user)
18
19    with open(fname, 'w') as f:
20      for page in Cursor(client.user_timeline, screen_name=user,
21                         count=200).pages(16):
22        for status in page:
23          if (status._json['text'].split(' ')[0] != 'RT'):
24            f.write(json.dumps(status._json)+"\n")
```

Figura 51. Fichero twitter_get_user_timeline.py

La estructura general de la aplicación es la misma que en el caso anterior. En primer lugar se realiza la autenticación en la API de Twitter, para posteriormente crear un fichero `user_timeline_nombre_usuario.jsonl`, en el que se irán almacenando todos los tweets publicados por el usuario, conforme se realizan peticiones a la API.

Una vez tenemos los ficheros JSON, que situaremos en el sistema de almacenamiento distribuido HDFS, de modo que sean fácilmente accesibles desde todos los workers y el máster de Spark, podemos comenzar con la aplicación Spark.

Como ya se ha comentado, la aplicación está estructurada en tres fases, cada una de las cuales se centra en realizar un análisis más específico del usuario. Existe una parte común a estas fases, que consiste básicamente en importar el contexto de Spark y configurarlo. Le ponemos un nombre a la aplicación y especificamos cuál es el master de Spark al que nos queremos conectar. También especificamos el nombre del usuario al que vamos a analizar.

```
In [1]: from pyspark import SparkContext
        from pyspark import SparkConf
        conf = SparkConf().setAppName("user").setMaster("spark://tfmspark.ddns.net:7077")
        sc = SparkContext.getOrCreate(conf=conf)

        Especificamos el nombre del usuario al que vamos a analizar.

In [3]: screen_name = 'CayeRodriguez'
```

Figura 52. Importar y configurar el contexto de Spark

Tras esto, ya podemos utilizar todas las funcionalidades de Spark.

6.1.1 Análisis de personas (seguidores, amigos, alcanzabilidad)

El primer análisis a realizar se corresponde a los seguidores y amigos de los usuarios especificados. Para ello, usaremos los ficheros `followers.jsonl` y `friends.jsonl`, donde se encuentran almacenados todos los seguidores y amigos, del usuario.

Antes de nada, resulta de interés explicar las relaciones de "amistad" que se establecen en la red social Twitter. Ya se ha introducido el concepto de amigos y seguidores, sin embargo, es más sencillo de entender con un simple grafo.

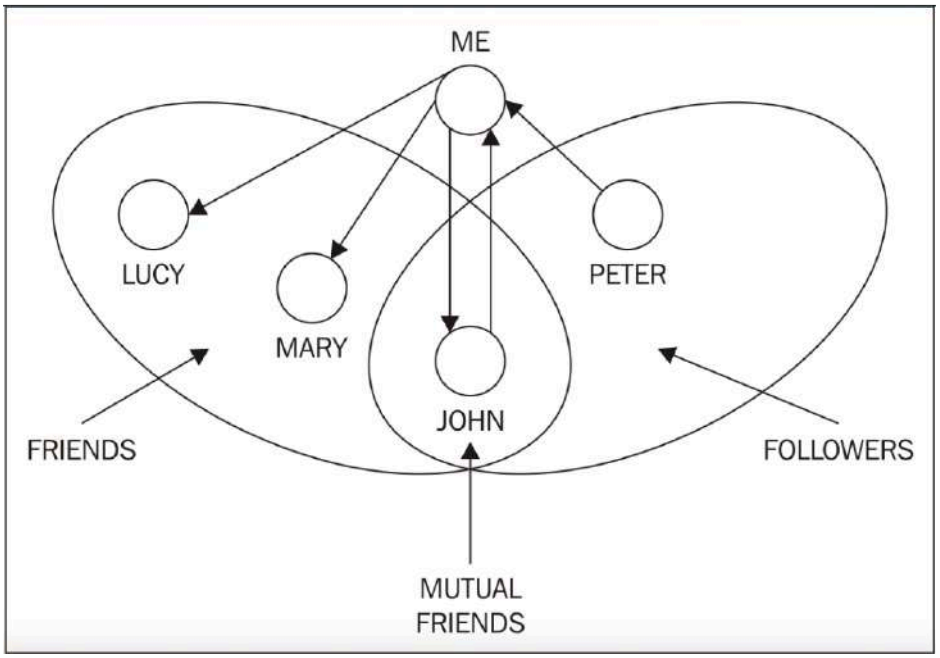

Figura 53. Twitter: amigos y seguidores

En la imagen anterior podemos observar las posibles relaciones establecidas entre usuarios de Twitter. En este ejemplo, estoy conectado con cuatro usuarios distintos: John y Peter me siguen, por tanto se denominan ***seguidores***. Por otro lado, yo sigo a Lucy, Mary y John, considerándose ***amigos***. John pertenece a ambos grupos, por lo que se considera que la relación es de ***amigos mutuos***. Otras relaciones básicas que se pueden extraer de aquí son los seguidores que no sigo, y los amigos que no me siguen. Trataremos de analizar todas estas relaciones con la aplicación Spark.

Calculamos el número de seguidores, la alcanzabilidad y el número medio de seguidores de cada seguidor. Para ello haremos uso del fichero `followers.jsonl` que hemos creado anteriormente, situado en el sistema HDFS.

```
In [17]: import json
         hdfs = 'hdfs://tfmhdfs.ddns.net:9000/user/centos/files/'
         followers_file = 'users/{}/followers.jsonl'.format(screen_name)

         reach = sc.textFile(hdfs + followers_file).map(lambda x: json.loads(x)['followers_count'])

In [18]: sum_reach = reach.sum()

         nfollowers = reach.count()

         avg_followers = round(sum_reach / nfollowers, 2)
```

Figura 54. Calcular seguidores, alcanzabilidad y número medio de seguidores de cada seguidor

En primer lugar creamos un RDD a partir del fichero de texto en formato JSON, utilizando la función de Spark **`textFile`**, al que realizaremos una transformación (**`map`**) para quedarnos únicamente con el campo `followers_count` de cada objeto *User* (este campo se corresponde con el número de seguidores de cada seguidor). De este modo, podremos calcular el número de seguidores realizando la operación **`count`** sobre el RDD, que cuenta el número de elementos contenidos en el RDD (uno por cada usuario en el fichero JSON). Por otro lado, calculamos el número medio de seguidores de los seguidores del usuario, para conocer la alcanzabilidad del mismo, es decir, a cuántas personas podría llegar si todos los seguidores retwittearan uno de sus tweets. Esto se puede calcular fácilmente dividiendo la suma total de seguidores de los seguidores (calculado a su vez realizando la operación **`sum`** sobre el RDD, que suma todos los valores numéricos de los elementos del RDD) entre el número de seguidores del usuario calculado previamente.

Por otro lado, calculamos el número de amigos, amigos mutuos, amigos que no están siguiendo y seguidores a los que no sigue el usuario.

```
In [19]: friends_file = 'users/{}/friends.jsonl'.format(screen_name)


         followers = sc.textFile(hdfs + followers_file).map(lambda x: json.loads(x)['screen_name'])
         friends = sc.textFile(hdfs + friends_file).map(lambda x: json.loads(x)['screen_name'])

         nfriends = friends.count()

         mutual = friends.intersection(followers).count()

         friends_not_following = nfriends - mutual

         followers_not_followed = nfollowers - mutual
```

Figura 55. Calcular número de amigos , amigos mutuos, amigos que no siguen al usuario y seguidores a los que no sigue el usuario

Creamos RDDs con los nombres de usuario de los seguidores y de los amigos del usuario. Calculamos 4 valores de interés:

- **Número de amigos**: realizando un `count` sobre el RDD que contiene los nombres de usuario de los amigos.
- **Amigos mutuos**: se trata de las personas a las que sigue el usuario y que a su vez siguen al usuario. Podemos calcularlo mediante la intersección de los dos RDDs, es decir, las personas que aparezcan en las dos listas (amigos y seguidores). Spark ofrece una operación de transformación, **`intersect`**, que hace esto mismo: devuelve un nuevo RDD que contenga los elementos comunes a ambos RDDs. Finalmente se realiza un `count` para quedarnos con el número.

- **Amigos que no siguen al usuario:** personas a las que sigue el usuario y que no le siguen. Una vez llegados a este punto, el cálculo es trivial. Se trata de la diferencia entre el número total de amigos y los amigos mutuos.
- **Seguidores a los que no sigue el usuario:** del mismo modo que el anterior, podemos obtenerlo con la diferencia entre el número total de seguidores y los amigos mutuos.

Finalmente, procedemos a la visualización de resultados.

```
In [20]: print("{} followers".format(nfollowers))
         print("{} friends".format(nfriends))
         print("{} mutual friends".format(mutual))
         print("{} friends are not following back".format(friends_not_following))
         print("{} followers are not followed back by {}".format(followers_not_followed, screen_name))
         print("{} users reached by 1-degree connections".format(sum_reach))
         print("Average number of followers for {}'s followers: {}".format(screen_name, avg_followers))

         126 followers
         246 friends
         124 mutual friends
         122 friends are not following back
         2 followers are not followed back by CayeRodriguez
         71506 users reached by 1-degree connections
         Average number of followers for CayeRodriguez's followers: 567.51
```

Figura 56. Análisis de personas: visualización de resultados

Más que el número de seguidores y amigos, lo más interesante que podemos extraer de estos resultados es la alcanzabilidad. Se podría pensar que un usuario con muchos seguidores es más interesante (en cuanto a temas de marketing y publicidad, por ejemplo) que uno con menos seguidores, y en parte es así, sin embargo, podría resultar de mayor interés un usuario cuya alcanzabilidad en un salto sea mayor, aunque tenga menos seguidores. Esto significa que, en media, los seguidores de ese usuario tienen más seguidores que uno con una alcanzabilidad menor y, por tanto, se podría conseguir una mayor visualización de los tweets publicados por este usuario, siempre y cuando sus seguidores directos interactúen activamente con él, retwitteando sus tweets o marcándolos como "me gusta". Esto es lo que analizaremos a continuación.

6.1.2 Análisis de influencia del usuario (retweets, favoritos)

Uno de los personajes de mayor interés y de los que más se habla en términos de redes sociales, es el *influencer*. Se trata típicamente de usuarios activos con su comunidad. En el caso concreto de Twitter, un influencer es un usuario que publica muchos tweets acerca de los temas que le interesan. Suelen estar bien conectados, ya que siguen y son seguidos por muchos usuarios, también involucrados en la comunidad.

Esto podría explicar el especial interés que tienen hoy en día estos influencers, en campañas de marketing o publicidad, donde podría alcanzarse a grandes cantidades de personas con los mismos intereses.

Otro concepto distinto, pero relacionado, es el compromiso (*engagement*). Se trata de la evaluación de la respuesta a una oferta particular. La medida del compromiso es algo muy importante, ya que ayuda a definir y entender las estrategias, con el fin de maximizar la interacción con las redes. En Twitter, el compromiso de un usuario se mide retwitteando o marcando como "me gusta" un tweet, dándole más visibilidad al mismo.

Como ya hemos adelantado, un número alto de seguidores se relaciona intuitivamente con alcanzar a un mayor número de personas, pero podría no ser del todo así. Trataremos de realizar un análisis referente a este aspecto, para tratar de medir la influencia de un usuario y el compromiso de sus seguidores.

En el fichero `user_timeline_nombre_usuario.jsonl` se encuentran todos los tweets del usuario, con lo cual, este será el fichero que utilizaremos para realizar el análisis. Simplemente contaremos el número de veces que los tweets del usuario han sido retwitteados y marcados como "me gusta", y realizaremos algunas operaciones básicas para obtener estadísticas de interés.

```
In [21]: timeline_file = 'user_timeline_{}.jsonl'.format(screen_name)

         timeline = sc.textFile(hdfs + timeline_file)
         favorite_count = timeline.map(lambda x: json.loads(x)['favorite_count']).sum()
         retweet_count = timeline.map(lambda x: json.loads(x)['retweet_count']).sum()

         ntweets = timeline.count()

         avg_favorite = round(favorite_count / ntweets, 2)
         avg_retweet = round(retweet_count / ntweets, 2)
         favorite_per_user = round(favorite_count / nfollowers, 2)
         retweet_per_user = round(retweet_count / nfollowers, 2)
```

Figura 57. Análisis de influencia de un usuario

Accedemos al contenido del fichero almacenado en el sistema HDFS mediante la función de Spark **`textFile`**. Crearemos dos RDDs de este fichero: uno para los retweets y otro para los "me gusta". Para ello, transformaremos el RDD inicial para quedarnos únicamente con el campo del tweet que nos interesa, mediante la operación de transformación **`map`**. En el caso de los "me gusta", tenemos el campo `favorite_count`, y para el caso de los retweets tenemos el campo `retweet_count`. Nos interesan los valores totales, no los valores individuales de cada tweet, por lo que realizaremos un **`sum`** sobre cada RDD para obtener el total de retweets y "me gusta", respectivamente. Por otro lado, contamos el número total de tweets publicados por el usuario ejecutando la acción **`count`** sobre el RDD inicial.

Una vez tenemos estos valores numéricos, calcularemos algunas estadísticas básicas:

- **Número medio de "me gusta" por tweet**: lo calculamos dividiendo el número total de "me gusta" entre el total de tweets publicados por el usuario.
- **Número medio de retweets por tweet**: del mismo modo, dividimos el número total de retweets entre el total de tweets del usuario.
- **Número medio de "me gusta" por usuario**: para calcular las estadísticas por usuario, aprovecharemos que ya conocemos el número de seguidores del usuario por el análisis anterior. Dividimos el número total de "me gusta" entre el número de seguidores del usuario.
- **Número medio de retweets por usuario**: al igual que el caso anterior, se calcula dividiendo el número total de retweets entre el número de seguidores.

Procedemos a visualizar los resultados obtenidos.

```
In [22]: print("Total number of tweets: {}".format(ntweets))
         print("Favorited {} times ({} per tweet, {} per user)".format(favorite_count, avg_favorite, favorite_per_user))
         print("Retweeted {} times ({} per tweet, {} per user)".format(retweet_count, avg_retweet, retweet_per_user))

         Total number of tweets: 2315
         Favorited 53 times (0.02 per tweet, 0.42 per user)
         Retweeted 248 times (0.11 per tweet, 1.97 per user)
```

Figura 58. Análisis de influencia: visualización de resultados

Las estadísticas ofrecen una gran cantidad de información en cuanto a la influencia de un usuario, así como del compromiso de sus seguidores. Se podría hacer uso de estos valores para tomar una decisión en cuanto a si un usuario interesa en mayor o menor medida para una campaña publicitaria concreta. Por supuesto, habría que tener en cuenta también los resultados obtenidos del análisis anterior, en cuanto al número de seguidores y la alcanzabilidad.

6.1.3 Análisis de términos más usados: Word Cloud

Ya que tenemos todos los tweets escritos por el usuario, podemos aprovechar para analizar las palabras más usadas por el mismo. Esto nos puede dar una idea de qué tipo de usuario se trata.

En primer lugar, vamos a realizar un filtrado del fichero, de modo que nos quedaremos únicamente con el campo `text` de cada tweet, que es el que contiene el texto en sí, y posteriormente eliminaremos enlaces y menciones, de modo que el conteo de palabras funcione correctamente. Para comprender perfectamente qué operaciones se van realizando sobre los RDDs, se mostrará en cada paso el contenido resultante, de modo que se irá viendo cómo vamos modificando el RDD para obtener lo que necesitamos.

```
In [28]: texts = timeline.map(lambda x: json.loads(x)['text'])
         texts.collect()

Out[28]: ['@Ismawel_mar estabas viendo la peli de telecinco o que? xD',
          'Receive your free dev stickers, worldwide! @notifuse https://t.co/id97p4LrSX',
          'Mejor escena de #AquiNoHayQuienViva las lentejas',
          'Y esto también @alerguez1994  https://t.co/jI6hnHE0ae',
          '@alerguez1994 bum',
          'Feliz año a todos! :)',
          '@Franxi_rios @MiguePepper  https://t.co/ZB94hxlCbT',
          '@sergiosucino @alerodri01 si se puede si somos amigos en Facebook',
          '@sergiosucino @alerodri01 no je ahora probamos',
          '@sergiosucino @alerodri01 es compatible el juego de android con el de iOS?',
          '@alerodri01 así me gusta ',
          '@alerodri01 que juego es ese? ',
          '@AnaPosu será que a ti te gusta levantarte a las 8 ',
          '@AnaPosu nada nada...que hay algunas que tienen tiempo para siestas..... ',
          '@AnaPosu como no...',
          'Feliz año a todos :)',
          'Me acabo de dar cuenta de que estaban echando piratas del caribe, harry potter y el señor de los anillos',
          'Piratas del caribe #bestfilmever',
          'Cómo prepararse una asignatura en 2 días - Tercera Parte',
```

Figura 59. Filtrado de texto

Como hemos hecho en los análisis anteriores, realizamos un **`map`** sobre el RDD inicial (obtenido a partir del fichero de texto), para quedarnos con un único campo del tweet, en este caso, `text`. Podemos observar que cada elemento del RDD se corresponde con una cadena de texto, con el contenido textual de cada tweet del usuario. Para visualizar el contenido del RDD se ha utilizado la acción **`collect`** de Spark. Ya se ha comentado que los RDD de Spark siguen el paradigma de *lazy evaluation*, es decir, que no se evalúan hasta que no se necesitan realmente. Al usar la función `collect`, hacemos que Spark ejecute las operaciones sobre el RDD y nos devuelva el contenido actual del mismo.

Tenemos que hacer limpieza en el texto de los tweets. Será interesante eliminar del texto la palabra RT que precede a cada tweet que haya sido retwitteado de otro usuario, además de las URLs y menciones a otros usuarios (precedidos de @), así como poner todo el texto en minúsculas o eliminar los elementos de puntuación.

```
In [32]: import string
         import re

         clean = texts.map(lambda x: x.lower())
                      .map(lambda x: re.sub(r"http\S+", "", x))
                      .map(lambda x: re.sub(r"@\S+", "", x))
                      .map(lambda x: re.sub(r"\\\S+", "", x))
                      .map(lambda x: "".join(c for c in x if c not in ('1','.', ',',':', ';', '?', '(', ')', '[', ']')))
                     .map(lambda x: ''.join(i for i in x if not i.isdigit()))

         clean.collect()

Out[32]: [' estabas viendo la peli de telecinco o que xd',
          'receive your free dev stickers worldwide  ',
          'mejor escena de #aquinohayquienviva las lentejas',
          'y esto también   ',
          ' bum',
          'feliz año a todos ',
          '   ',
          '  si se puede si somos amigos en facebook',
          '  no je ahora probamos',
          '  es compatible el juego de android con el de ios',
          ' así me gusta 👍🏻',
          ' que juego es ese 😴',
          ' será que a ti te gusta levantarte a las 😴😴',
          ' nada nadaque hay algunas que tienen tiempo para siestas😴👀',
          ' como no',
          'feliz año a todos ',
          'me acabo de dar cuenta de que estaban echando piratas del caribe harry potter y el señor de los anillos',
          'piratas del caribe #bestfilmever',
          'cómo prepararse una asignatura en  días - tercera parte',
          ' \n',
```

Figura 60. Limpieza de texto

Realizamos una serie de transformaciones sucesivas, con `map`, al RDD con el texto de los tweets. Así, ponemos todo el texto en minúscula, eliminamos enlaces y menciones a otros usuarios, eliminamos también la puntuación y los dígitos numéricos. Con esto, obtenemos el texto en limpio de todos los tweets publicados por el usuario.

El siguiente paso será separar todas las palabras. Tendremos un elemento en el RDD por cada palabra de cada tweet. Así será mucho más fácil realizar el recuento de palabras.

```
In [33]: words = clean.flatMap(lambda x: x.split(' '))
         words.collect()

Out[33]: ['',
          'estabas',
          'viendo',
          'la',
          'peli',
          'de',
          'telecinco',
          'o',
          'que',
          'xd',
          'receive',
          'your',
          'free',
          'dev',
          'stickers',
          'worldwide',
          '',
          '',
          'mejor',
```

Figura 61. Separación del texto en palabras individuales

En este caso hemos realizado la transformación mediante la operación **`flatmap`**. Es una operación similar a `map`, que permite devolver más de un elemento. La función `map` realiza una transformación sobre cada elemento del RDD, devolviendo un nuevo RDD con el resultado, mientras que `flatmap` permite mapear cada elemento a 0 o más salidas, de modo que podemos transformar un elemento en varios de una forma muy simple. En la salida podemos observar que cada elemento es una única palabra.

Lo siguiente que haremos será crear un RDD de pares, añadiendo un 1 a cada palabra. Para ello usaremos la función `map` de Spark, que ejecuta una función sobre cada elemento del RDD.

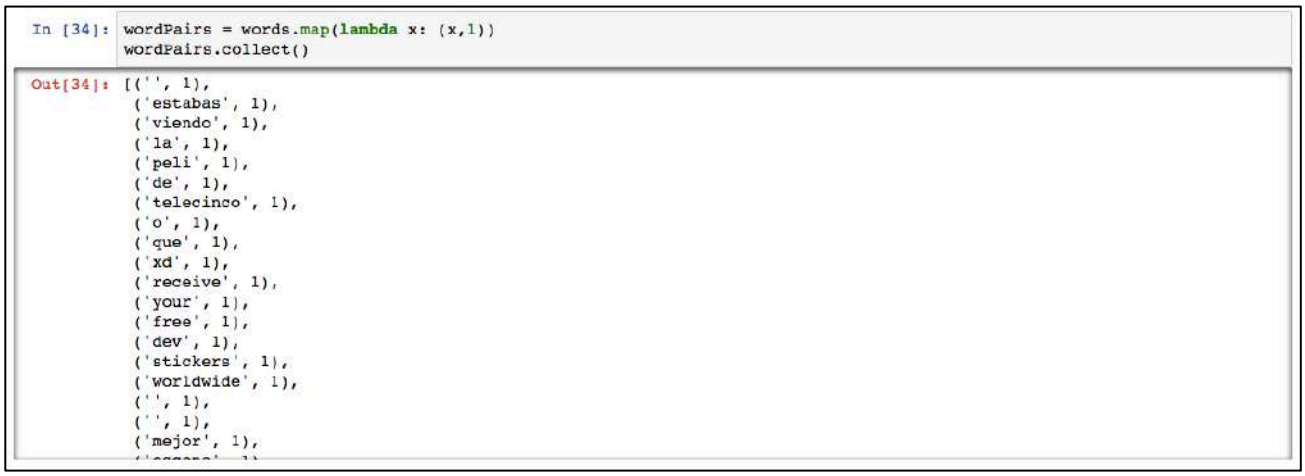

Figura 62. Creación de RDD de pares

Simplemente hemos convertido cada elemento del RDD en un par palabra-1. Esto nos permitirá realizar contar las palabras de forma inmediata, como veremos a continuación.

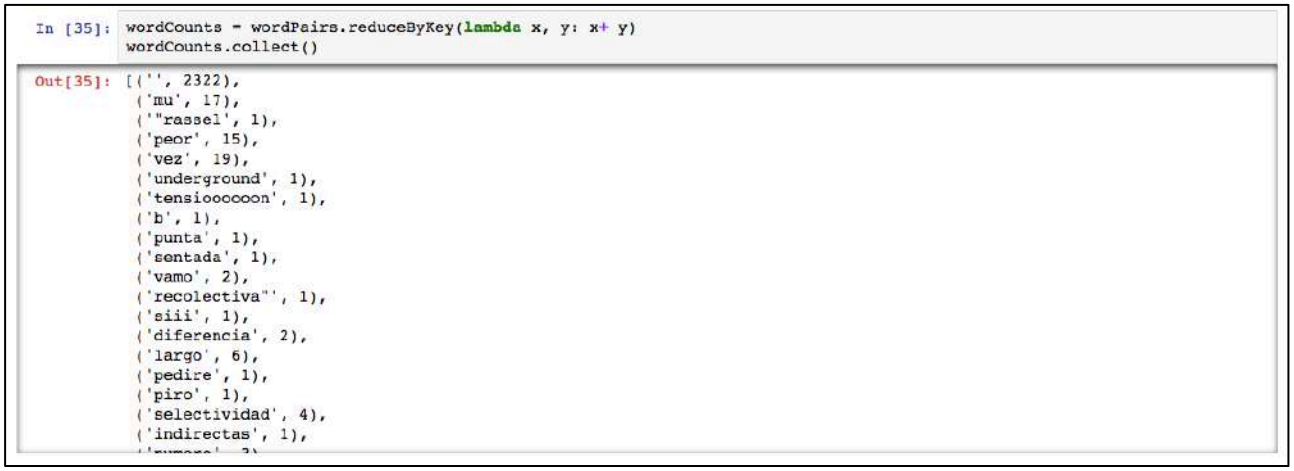

Figura 63. Frecuencia de cada palabra

Finalmente, procedemos a contar la frecuencia de cada palabra. Esto lo hacemos utilizando una función de reducción de Spark: **`reduceByKey`**. Se agrupan todos los elementos del RDD que tengan la misma clave (la palabra) y se sumarán todos los 1s, de modo que obtendremos un RDD con todas las palabras junto a su frecuencia de aparición.

Ahora, transformamos el RDD resultante en un diccionario de Python, y eliminamos algunos elementos que pueden distorsionar el resultado, como los elementos vacíos. También se ha decidido eliminar algunas palabras de uso común, de modo que el resultado sea de mayor utilidad.

```
In [36]: wordCountsDict = wordCounts.collectAsMap()
         del (wordCountsDict['']])
         del (wordCountsDict['"']])
         del (wordCountsDict['-'])

         delete_list = ["rt", "que", "de", "a", "y", "no", "si", "el", "la", "los", "las", "un", "una", "unos", "unas",
                        "en", "lo", "es", "ya", "por", "yo", "tu", "me", "se", "te", "con", "mi", "para", "he", "ha", "o",
                        "al", "pa", "hay", "del", "mas", "más", "como", "has", "ni", "sin", "eso", "le", "este", "esta",
                        "hace", "hacer", "pero", "hasta", "va", "porque", "pues", "menos", "han", "to", "tiene", "asi", "así",
                        "aquí", "algo", "x", "ir", "son", "estoy", "voy", "nos", "muy", "mucho", "luego", "estos", "estas"]
         for word in delete_list:
             del (wordCountsDict[word])
```

Figura 64. Conversión a diccionario y supresión de elementos distorsionadores

Para realizar la conversión a diccionario de Python, Spark ofrece la función **`collectAsMap`**, que devuelve los pares clave-valor del RDD al máster de Spark en forma de diccionario. Una vez tenemos el diccionario, simplemente eliminamos los elementos que pueden distorsionar el resultado, como los elementos vacíos (espacios), comillas dobles o guiones. También hay muchas palabras de uso común, que pueden enturbiar los resultados, por lo que eliminaremos algunas de estas palabras.

Nos interesa ordenar todas las palabras por su frecuencia, sin embargo, los diccionarios no se pueden ordenar (por definición). Entonces, usaremos la función `sorted` para convertir el diccionario en una lista ordenada según la frecuencia, y en orden descendente. El interés es debido a que usaremos una librería específica para visualizar el resultado, y requiere que se le pase la lista de palabras ordenada según su frecuencia de aparición.

```
In [37]: sortedList = sorted(wordCountsDict.items(), key=lambda x: x[1], reverse=True)
         sortedList

Out[37]: [('xd', 224),
          ('mañana', 116),
          ('tengo', 84),
          ('hoy', 63),
          ('buenas', 62),
          ('ahora', 56),
          ('examen', 54),
          ('noches', 53),
          ('cuando', 47),
          ('vamos', 47),
          ('ver', 46),
          ('hora', 45),
          ('vídeo', 42),
          ('nada', 42),
          ('todo', 40),
          ('bien', 39),
          ('u_u', 38),
          ('estudiar', 35),
          ('#twitteroff', 32),
```

Figura 65. Lista de palabras ordenada según frecuencia

Se puede apreciar que las palabras ya están ordenadas según su frecuencia, con lo cual podemos proceder a la visualización de resultados. Aquí ya tenemos la lista completa de palabras más usadas por el usuario en sus tweets, sin embargo, la forma de visualización no es cómoda. Por este motivo, se pensó en realizar un ***word cloud*** (nube de palabras), que se usa bastante para la visualización de este tipo de datos.

Un word cloud no es más que una nube de palabras en la que el tamaño de cada palabra se corresponde con su frecuencia, es decir, que las palabras de mayor tamaño serán las más utilizadas por el usuario. Para obtener el word cloud, vamos a utilizar la librería `wordcloud`.

```
In [38]: from wordcloud import WordCloud, STOPWORDS
```

Figura 66. Librería usada para crear el word cloud

Para generar el word cloud, le pasamos a la función la lista ordenada de palabras.

```
In [39]: # Generate a word cloud image
         wordcloud = WordCloud(max_font_size=40).generate_from_frequencies(sortedList)
```

Figura 67. Generación del word cloud

Por último, imprimimos la imagen con el resultado.

```
In [40]: import matplotlib.pyplot as plt
         plt.figure()
         plt.imshow(wordcloud, interpolation='bilinear')
         plt.axis("off")
         plt.show()
```

Figura 68. Código para imprimir el word cloud

Se adjunta a continuación el word cloud obtenido para el usuario analizado.

Figura 69. Word Cloud

Si analizamos este word cloud, podría deducirse que el usuario es un estudiante, debido a la aparición de palabras como *examen* o *estudiar*, entre las palabras más usadas. Con esto se pretende ofrecer una idea general de los temas que más interesan al usuario, obteniendo así una primera impresión del mismo.

Mediante la combinación de los tres análisis realizados al usuario, se pretende obtener, en líneas generales, el perfil del usuario, en términos de influencia, alcanzabilidad e intereses principales.

6.2 Análisis de tendencias

La segunda de las aplicaciones desarrolladas se centra en el módulo de Spark: Spark SQL. Gracias a este módulo se puede aprovechar la estructura de los datos utilizados (JSON es un formato semiestructurado) para trabajar con ellos como si se tratase de tablas de una base de datos relacional. Lo habitual es utilizar bases de datos no

relacionales cuando nos encontramos con una gran cantidad de datos, sin embargo, podemos aprovechar las propiedades de las bases de datos relacionales para extraer información útil de los datos de una forma mucho más sencilla.

En este caso, analizaremos otro de los conceptos más conocidos de Twitter: las tendencias. Las tendencias de Twitter son los temas más comentados por los usuarios en el momento en la red social. El objetivo de esta aplicación consistirá en descubrir cómo visualizar las tendencias según la ubicación, para posteriormente comparar las tendencias en España y las tendencias mundiales.

Lo primero que hay que hacer es importar el contexto de Spark y configurarlo. Le ponemos un nombre a la aplicación y especificamos cuál es el master de Spark al que nos queremos conectar.

```
In [2]: from pyspark import SparkContext
        from pyspark import SparkConf
        conf = SparkConf().setAppName("trends").setMaster("spark://tfmspark.ddns.net:7077")
        sc = SparkContext.getOrCreate(conf=conf)
```

Figura 70. Importar y configurar el contexto de Spark

A continuación necesitamos realizar la autenticación en la API de Twitter, para poder hacer uso de la API programáticamente.

```
In [4]: import os
        import sys
        from tweepy import API
        from tweepy import OAuthHandler

        Autenticación en la API de Twitter

In [5]: # Variables that contains the user credentials to access Twitter API
        ACCESS_TOKEN = 'XXXXXXXXXXXXXXX'
        ACCESS_SECRET = 'XXXXXXXXXXXXXXX'
        CONSUMER_KEY = 'XXXXXXXXXXXXX'
        CONSUMER_SECRET = 'XXXXXXXXXXXXXXX'

In [6]: auth = OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)
        auth.set_access_token(ACCESS_TOKEN, ACCESS_SECRET)

In [7]: client = API(auth)
```

Figura 71. Autenticación en la API de Twitter

Existe una función de la API de Twitter que nos permite obtener la lista de posibles localizaciones para las que se puede pedir las tendencias. Haremos uso de esta función para crear un RDD con el contenido de la respuesta.

```
In [8]: trends = sc.parallelize(client.trends_available())

In [9]: trends.collect()

Out[9]: [{'country': '',
          'countryCode': None,
          'name': 'Worldwide',
          'parentid': 0,
          'placeType': {'code': 19, 'name': 'Supername'},
          'url': 'http://where.yahooapis.com/v1/place/1',
          'woeid': 1},
         {'country': 'Canada',
          'countryCode': 'CA',
          'name': 'Winnipeg',
          'parentid': 23424775,
          'placeType': {'code': 7, 'name': 'Town'},
          'url': 'http://where.yahooapis.com/v1/place/2972',
          'woeid': 2972},
         {'country': 'Canada',
          'countryCode': 'CA',
          'name': 'Ottawa',
          'parentid': 23424775,
          'placeType': {'code': 7, 'name': 'Town'},
```

Figura 72. RDD con la lista de localizaciones para tendencias

Para crear el RDD se ha utilizado la función de Spark **`parallelize`**, que genera el RDD, directamente, a partir del objeto JSON con las tendencias. Visualizamos el contenido del mismo usando **`collect`**, como en casos anteriores.

A la vista de la respuesta, podemos intuir que cada localización tiene un identificador (`woeid`) que podremos utilizar para pedir la lista de tendencias de dicho lugar. Sin embargo, es una tarea tediosa tener que buscar en dicha respuesta la localización concreta que necesitemos, ya que no se sigue ningún tipo de orden a simple vista.

Vamos a utilizar Spark SQL para solucionar este problema. Tendremos que obtener una sesión de Spark SQL, y crearemos un DataFrame a partir del RDD anterior, con el contenido en formato JSON de las localizaciones de las tendencias.

```
In [10]: from pyspark.sql import SparkSession
         spark = SparkSession.builder.appName("Spark SQL trends").getOrCreate()

         Creamos un DataFrame a partir del RDD anterior, que contiene el JSON con todos los países.

In [11]: df = spark.read.json(trends)
```

Figura 73. Creación de la sesión de Spark SQL y el DataFrame

Creamos el DataFrame utilizando la función **`read`**, especificando el formato de los datos, **`json`**, a partir del RDD anterior. Ahora ya podemos usar las funcionalidades de Spark SQL para aprovechar su potencial.

```
In [12]: df.show()
```

```
+--------------------+--------------+-----------+-----------+--------+---------+--------------------+-----+
|     _corrupt_record|       country|countryCode|       name|parentid|placeType|                 url|woeid|
+--------------------+--------------+-----------+-----------+--------+---------+--------------------+-----+
|{'parentid': 0, '...|          null|       null|       null|    null|     null|                null| null|
|                null|        Canada|         CA|   Winnipeg|23424775| [7,Town]|http://where.yaho...| 2972|
|                null|        Canada|         CA|     Ottawa|23424775| [7,Town]|http://where.yaho...| 3369|
|                null|        Canada|         CA|     Quebec|23424775| [7,Town]|http://where.yaho...| 3444|
|                null|        Canada|         CA|   Montreal|23424775| [7,Town]|http://where.yaho...| 3534|
|                null|        Canada|         CA|    Toronto|23424775| [7,Town]|http://where.yaho...| 4118|
|                null|        Canada|         CA|   Edmonton|23424775| [7,Town]|http://where.yaho...| 8676|
|                null|        Canada|         CA|    Calgary|23424775| [7,Town]|http://where.yaho...| 8775|
|                null|        Canada|         CA|  Vancouver|23424775| [7,Town]|http://where.yaho...| 9807|
|                null|United Kingdom|         GB| Birmingham|23424975| [7,Town]|http://where.yaho...|12723|
|                null|United Kingdom|         GB|  Blackpool|23424975| [7,Town]|http://where.yaho...|12903|
|                null|United Kingdom|         GB|Bournemouth|23424975| [7,Town]|http://where.yaho...|13383|
|                null|United Kingdom|         GB|   Brighton|23424975| [7,Town]|http://where.yaho...|13911|
|                null|United Kingdom|         GB|    Bristol|23424975| [7,Town]|http://where.yaho...|13963|
|                null|United Kingdom|         GB|    Cardiff|23424975| [7,Town]|http://where.yaho...|15127|
|                null|United Kingdom|         GB|   Coventry|23424975| [7,Town]|http://where.yaho...|17044|
|                null|United Kingdom|         GB|      Derby|23424975| [7,Town]|http://where.yaho...|18114|
|                null|United Kingdom|         GB|  Edinburgh|23424975| [7,Town]|http://where.yaho...|19344|
|                null|United Kingdom|         GB|    Glasgow|23424975| [7,Town]|http://where.yaho...|21125|
|                null|United Kingdom|         GB|       Hull|23424975| [7,Town]|http://where.yaho...|25211|
+--------------------+--------------+-----------+-----------+--------+---------+--------------------+-----+
only showing top 20 rows
```

Figura 74. Contenido del DataFrame en formato de tabla

La función **`show`** del DataFrame nos permite visualizar su contenido en formato de tabla, que es mucho más cómodo y eficiente. Spark SQL también permite inferir el esquema de la tabla de los datos, con la función **`printSchema`**.

```
In [13]: df.printSchema()
```

```
root
 |-- _corrupt_record: string (nullable = true)
 |-- country: string (nullable = true)
 |-- countryCode: string (nullable = true)
 |-- name: string (nullable = true)
 |-- parentid: long (nullable = true)
 |-- placeType: struct (nullable = true)
 |    |-- code: long (nullable = true)
 |    |-- name: string (nullable = true)
 |-- url: string (nullable = true)
 |-- woeid: long (nullable = true)
```

Figura 75. Esquema de los datos de la tabla

Podemos ver que, además de la estructura de los campos, también se muestra el tipo de cada campo de datos, que se ha inferido automáticamente analizando los datos contenidos en el DataFrame.

Una vez conocida la estructura y el esquema de los datos, podemos aplicar filtros y realizar consultas al DataFrame, como si se tratara de una tabla de una base de datos relacional. Por ejemplo, filtramos para quedarnos con los campos `name` y `woeid`, de todas las entradas cuyo país (`country`) sea España.

```
In [14]: df.select("name", "woeid").filter(df['country']=='Spain').show()
+----------+--------+
|      name|   woeid|
+----------+--------+
| Barcelona|  753692|
|    Bilbao|  754542|
|Las Palmas|  764814|
|    Madrid|  766273|
|    Malaga|  766356|
|    Murcia|  768026|
|     Palma|  769293|
|   Seville|  774508|
|  Valencia|  776688|
|  Zaragoza|  779063|
|     Spain|23424950|
+----------+--------+
```

Figura 76. Filtrado de localizaciones disponibles para tendencias en España

El comando utilizado se parece mucho a una sentencia SQL. Usamos **`select`** para quedarnos únicamente con los campos de la tabla que nos interesan, y utilizamos **`filter`** para aplicar un filtrado a las entradas de la tabla. Finalmente usamos **`show`** para ejecutar la sentencia y mostrar la salida.

En esta lista están todas las ciudades españolas para las que es posible obtener las tendencias actuales. Podríamos realizar una comparativa entre estas ciudades españolas, pero se ha preferido comparar las tendencias españolas, a nivel global, con las tendencias mundiales.

Obtenemos las tendencias españolas usando la API de Twitter con el identificador que vemos en la tabla anterior.

```
In [15]: spain_trends = client.trends_place(23424950)
```

Figura 77. Obtención de las tendencias españolas

De nuevo, queremos crear un DataFrame con estos datos. Anteriormente, se ha mostrado cómo crear un DataFrame a partir de un RDD. Ahora se mostrará que también es posible crearlo a partir de un fichero de texto.

```
In [28]: import json
         with open('spain_trends.json', 'w') as outfile:
             json.dump(spain_trends[0]['trends'], outfile)

         Creamos el DataFrame a partir del fichero .json.

In [97]: df_spain = spark.read.json('spain_trends.json')
```

Figura 78. Creación de DataFrame desde fichero de texto

Se ha usado la misma función que en el caso anterior, aunque ahora le hemos pasado un fichero JSON en lugar del RDD. Visualizamos el contenido del DataFrame usando **`show`**.

```
In [98]: df_spain.show()
```

name	promoted_content	query	tweet_volume	url
#HayQueEcharlos	null	%23HayQueEcharlos	44849	http://twitter.co...
#VivaLaVida1	null	%23VivaLaVida1	null	http://twitter.co...
Dumoulin	null	Dumoulin	22874	http://twitter.co...
Mc Men	null	%22Mc+Men%22	null	http://twitter.co...
Rubén Castro	null	%22Rub%C3%A9n+Cas...	null	http://twitter.co...
#TJCamisetaGuay	null	%23TJCamisetaGuay	null	http://twitter.co...
Hamburgo	null	Hamburgo	null	http://twitter.co...
Xabi Alonso	null	%22Xabi+Alonso%22	38118	http://twitter.co...
#CuatroAlAbordaje	null	%23CuatroAlAbordaje	null	http://twitter.co...
#LigaIberdrola	null	%23LigaIberdrola	11785	http://twitter.co...
Svitolina	null	Svitolina	null	http://twitter.co...
Le Mans	null	%22Le+Mans%22	null	http://twitter.co...
Pippa Middleton	null	%22Pippa+Middleto...	59089	http://twitter.co...
Igor Antón	null	%22Igor+Ant%C3%B3...	null	http://twitter.co...
Bartra	null	Bartra	null	http://twitter.co...
Nibali	null	Nibali	null	http://twitter.co...
Arabia Saudí	null	%22Arabia+Saud%C3...	null	http://twitter.co...
Hoffenheim	null	Hoffenheim	null	http://twitter.co...
Bremen	null	Bremen	null	http://twitter.co...
Lahm	null	Lahm	60112	http://twitter.co...

only showing top 20 rows

Figura 79. Tabla con tendencias españolas

Nuevamente, inferimos el esquema.

```
In [99]: df_spain.printSchema()
```

```
root
 |-- name: string (nullable = true)
 |-- promoted_content: string (nullable = true)
 |-- query: string (nullable = true)
 |-- tweet_volume: long (nullable = true)
 |-- url: string (nullable = true)
```

Figura 80. Esquema de la tabla de tendencias españolas

Finalmente, filtramos únicamente la columna que contiene el nombre de las tendencias, y nos quedamos con las 10 primeras.

```
In [108]: df_spain.select('name').show(10)
```

name
#HayQueEcharlos
#VivaLaVida1
Dumoulin
Mc Men
Rubén Castro
#TJCamisetaGuay
Hamburgo
Xabi Alonso
#CuatroAlAbordaje
#LigaIberdrola

only showing top 10 rows

Figura 81. Lista con las 10 mayores tendencias en España

Para terminar, compararemos con las tendencias mundiales, cuyo identificador es el 1, así que repetimos todo el proceso con este nuevo identificador.

```
In [109]: world_trends = client.trends_place(1)

In [111]: with open('world_trends.json', 'w') as outfile:
              json.dump(world_trends[0]['trends'], outfile)

In [112]: df_world = spark.read.json('world_trends.json')

In [113]: df_world.select('name').show(10)
+----------------+
|            name|
+----------------+
|      #بنت_تر|ند|
|#お前らガチ泣きしたシーン晒せよ|
|     #Bundesliga|
|       #lovelive|
|#NamoreAlguémQue|
|        Millwall|
|        Dumoulin|
|        Dortmund|
|       NOT TODAY|
|     Enes Kanter|
+----------------+
only showing top 10 rows
```

Figura 82. Lista de tendencias mundiales

Podemos observar que en la lista de tendencias mundiales aparecen tendencias en varios idiomas (árabe, japonés, inglés…). Si nos fijamos y comparamos ambos resultados, vemos que hay una tendencia común (aunque en distinta posición): Dumoulin, un ciclista que ha ganado una etapa del Giro de Italia.

6.3 Streaming: Frecuencia de tweets

Con esta aplicación se pretende hacer uso del módulo Spark Streaming de Apache Spark para analizar la frecuencia de tweets sobre un tema determinado durante un intervalo dado, así como los usuarios más activos durante este periodo. Para llevarla a cabo, tendremos que desplegar un escenario diferente al de los casos anteriores, ya que Spark Streaming ingiere los datos de fuentes más avanzadas, como Kafka, Flume o Kinesis. En nuestro caso, utilizaremos Apache Kafka para enviar los datos a Spark Streaming. Por este motivo, el escenario ha de incluir un servidor de Kafka y ZooKeeper, en el que se ejecute una aplicación que obtenga los tweets en streaming de Twitter y los envíe a Kafka para su posterior lectura en Spark. El escenario quedaría de la siguiente forma.

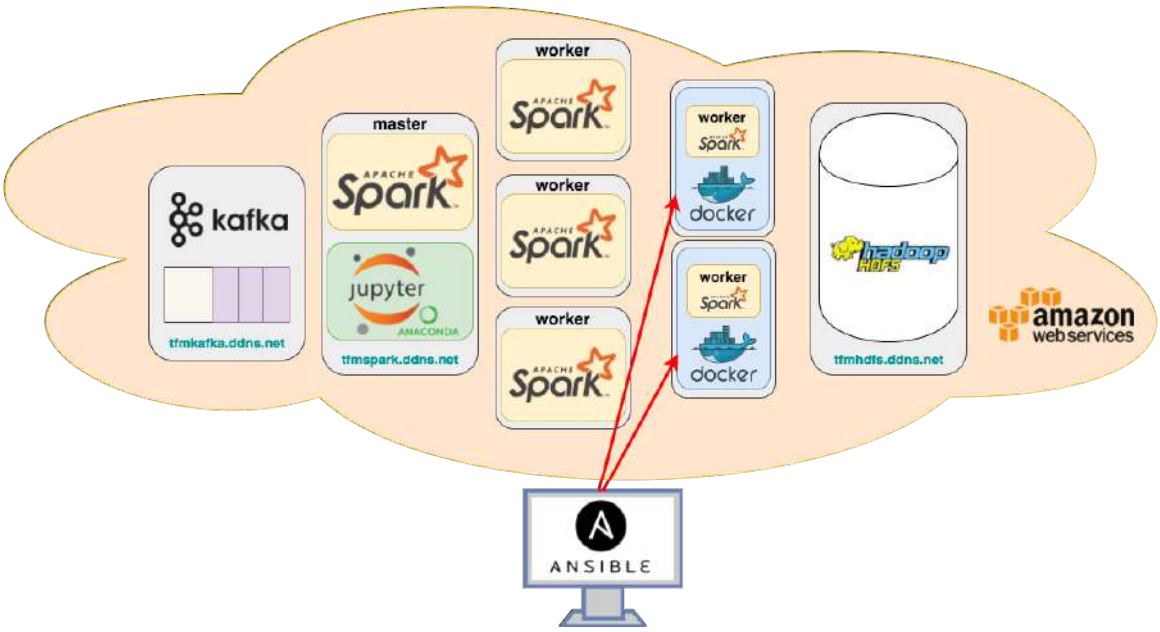

Figura 83. Escenario completo con servidor Kafka

Se ha utilizado el DNS dinámico de nuevo para este servidor Kafka, de modo que se pueda acceder al mismo de una manera mucho más sencilla a través del nombre `tfmkafka.ddns.net`.

6.3.1 Servidor Kafka

Lo primero que tenemos que hacer para poder llevar a cabo esta aplicación es desplegar un servidor de Apache Kafka. El primer paso consistirá en descargar e instalar Kafka:

```
curl -O dir_descarga_kafka0.8.2.1

tar -xzvf Kafka_2.1.0-0.8.2.1.tgz
```

Instalaremos la versión 0.8.2.1 de Kafka (el motivo se explicará más adelante). La dirección de descarga podemos obtenerla de la página oficial de Kafka: https://kafka.apache.org/downloads. El enlace de descarga usado es el siguiente: http://apache.rediris.es/kafka/0.8.2.1/kafka_2.10-0.8.2.1.tgz.

Una vez hecho esto, tenemos Kafka instalado en nuestra máquina, así como ZooKeeper, que es una dependencia de Kafka. Lo que nos queda por hacer es configurar el servidor con la dirección IP de la misma. Tenemos que hacer esto porque

se está realizando el despliegue en una instancia EC2 de Amazon Web Services, y de no hacerlo, el servidor no sería accesible a través de Internet.

Nos basta con modificar el fichero **config/server.properties** de Kafka. Tendremos que añadir la dirección pública (o el nombre DNS) de la instancia en la línea siguiente:

```
zookeeper.connect=dirección_pública:2181
```

Esta línea especifica dónde se encuentra el servidor de ZooKeeper al que debe conectarse el servidor Kafka.

Cabe destacar que si estamos realizando la instalación de Kafka en una máquina distinta al servidor de Spark (recomendable, debido a la memoria que necesita Spark para funcionar correctamente), tendremos también que instalar y configurar Java como lo hicimos anteriormente.

Por último, para arrancar el servidor Kafka, primero habrá que arrancar el servidor ZooKeeper:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

Y el servidor Kafka:

```
bin/kafka-server-start.sh config/server.properties
```

Si todo ha ido bien, podremos ver en ambas terminales el establecimiento de conexión entre el servidor Kafka y el servidor ZooKeeper.

Como comentario final, recordar que si se está llevando a cabo el despliegue de la infraestructura en un entorno real, habría que configurar adecuadamente los grupos de seguridad de la instancia EC2 de AWS, de modo que se permita el tráfico hacia los puertos de ZooKeeper (2181) y Kafka (9092).

6.3.2 Aplicación Twitter-Kafka

Se ha desarrollado una aplicación en Python que recibe tweets de Twitter usando la librería **tweepy**, y que los inyecta en Kafka utilizando la librería **pykafka**. El código de dicha aplicación se adjunta a continuación.

twitter-kafka.py

```python
import tweepy

from pykafka import KafkaClient

# Twitter API tokens
ACCESS_TOKEN = 'XXXXXXXXXXXXXXXX'
ACCESS_SECRET = 'XXXXXXXXXXXXXXXX'
CONSUMER_KEY = 'XXXXXXXXXXXXXXXX'
CONSUMER_SECRET = 'XXXXXXXXXXXXXXXX'

# Listener class to be used by tweepy
class StdOutListener(tweepy.StreamListener):

    # When a tweet comes, send attribute 'created_at' and 'screen_name' to Kafka
    def on_status(self, status):
        date = str(status.created_at)
        user = status._json['user']['screen_name']
        msg = date + "|" + user
        with mytopic.get_sync_producer() as producer:
            producer.produce(msg)
        return True

    def on_error(self, status_code):
        print('Got an error with status code: ' + str(status_code))
        return True

    def on_timeout(self):
        print('Timeout...')
        return True

# Main function
if __name__ == '__main__':

    listener = StdOutListener()

    oauth = tweepy.OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)
    oauth.set_access_token(ACCESS_TOKEN, ACCESS_SECRET)

    # Create twitter stream using OAuth credentials and the listener
    stream = tweepy.Stream(oauth, listener)

    # We need to specify the Kafka server
    client = KafkaClient(hosts="tfmkafka.ddns.net:9092")

    # This is the topic in the Kafka broker
    mytopic = client.topics['twitter']

    # Here we can filter tweets by any word
    stream.filter(track=['#RussianGP'])
```

Figura 84. Código aplicación twitter-kafka.py

En esta aplicación hemos tenido que utilizar las credenciales de OAuth de la API de Twitter para poder conectarnos a la misma y obtener el streaming de tweets. En el código hay que especificar la dirección del servidor Kafka al que debe enviar los datos. Podemos modificar el filtro de los tweets en la última línea, donde podemos añadir varias palabras de filtrado para las que se buscarán los tweets. Cuando los tweets llegan a la aplicación, enviamos a Kafka una cadena de texto formada por dos atributos extraídos de los tweets: el atributo **`created_at`**, que especifica la fecha de creación del

tweet, y el atributo **`screen_name`**, dentro del atributo **`user`** del tweet, que especifica el nombre del usuario que ha publicado el tweet. Nos basta con estos dos atributos para poder registrar la frecuencia de tweets durante el periodo del streaming y para buscar a los usuarios más activos.

Para poder escuchar un streaming de tweets usando la librería tweepy, tenemos que crear un `listener`. Este `listener` se encargará de estar continuamente a la espera de que lleguen nuevos tweets, y ejecutará el código que se incluya en la función `on_status`. En nuestro caso, extraeremos los atributos que nos interesan de cada tweet, crearemos la cadena de texto y crearemos un productor Kafka para enviar dicho mensaje. También habrá que definir las funciones `on_error` y `on_timeout`, para especificar los casos de error y timeout.

A continuación se genera el flujo de tweets, especificando las credenciales OAuth de Twitter y el listener. Es posible filtrar el flujo por alguna palabra o término de interés, o incluso varios. En el caso de nuestra aplicación, vamos a analizar el streaming durante una carrera de Fórmula 1, para poder recolectar una gran cantidad de tweets y visualizar las variaciones en la frecuencia de los mismos durante todo el periodo. Por este motivo, filtraremos por el hashtag **`#RussianGP`**, que será el utilizado por los aficionados a la Fórmula 1 para esta carrera.

Por último, nos queda definir el cliente Kafka que usará la aplicación para enviar los mensajes. Debemos especificar la dirección del servidor Kafka (IP y puerto), así como el topic Kafka que se usará para enviar los mensajes y que a su vez usará Spark para consumirlos.

6.3.3 Integración de Spark Streaming y Apache Kafka

Para poder usar Kafka como fuente de datos de Spark Streaming, necesitamos un paquete adicional en Spark. Existen dos paquetes de Kafka disponibles para Spark Streaming: uno para la versión 0.8 de Kafka y otro para la versión 0.10. La segunda de ellas se encuentra en estado experimental para el lenguaje Python, que es el que hemos estado usando, con lo cual nos quedaremos con el primer paquete (compatible con Kafka 0.8.2.1 o versiones superiores). A continuación se detallarán los pasos necesarios para configurar Spark Streaming para poder recibir datos de Kafka.

Lo primero que tenemos que hacer es asegurarnos de que los paquetes que vamos a utilizar están disponibles para Spark. En lugar de descargar los ficheros `jar` y tener que preocuparnos de las rutas, podemos usar la opción **`--packages`** para especificar el paquete de *maven* (a partir de `group/artifact/version`), y Spark se encargará de descargarlo automáticamente. Para hacer esto, podemos modificar la variable de

entorno **PYSPARK_SUBMIT_ARGS**. Como hemos hecho en otras ocasiones, añadiremos una nueva línea al fichero `.bashrc` exportando esta variable.

```
export PYSPARK_SUBMIT_ARGS="--packages org.apache.spark:spark-streaming-kafka-0-8_2.11:2.1.0 pyspark-shell"
```

Para que los cambios tengan efecto:

```
source .bashrc
```

De este modo, cuando ejecutemos la aplicación desde Jupyter, Spark descargará el paquete si no lo tiene disponible.

6.3.4 Aplicación Spark Streaming

Una vez tenemos todo el escenario completo y funcionando, ya podemos pasar a analizar el streaming de Twitter. Como se ha ido adelantando, utilizaremos el módulo Spark Streaming junto con el motor de Spark para analizar un flujo de tweets durante un intervalo de tiempo determinado, que en nuestro caso será una carrera automovilística.

Al igual que en las aplicaciones anteriores, el primer paso es importar y configurar el contexto de Spark.

```
In [1]: from pyspark import SparkContext
        from pyspark import SparkConf
        conf = SparkConf().setAppName("streaming").setMaster("spark://tfmspark.ddns.net:7077")
        sc = SparkContext.getOrCreate(conf=conf)
```

Figura 85. Importar y configurar el contexto de Spark

Para poder utilizar el módulo de Spark Streaming con Kafka es necesario importar más librerías.

```
In [2]: # Spark Streaming
        from pyspark.streaming import StreamingContext
        # Kafka
        from pyspark.streaming.kafka import KafkaUtils
```

Figura 86. Importar librerías para usar Spark Streaming con Kafka

A continuación crearemos el contexto de Spark Streaming. Para hacer esto tenemos que pasarle el contexto de Spark y especificar un intervalo de lote, es decir, el tiempo que esperará Spark Streaming para mirar si hay nuevos mensajes en la cola Kafka.

```
In [3]: ssc = StreamingContext(sc, 60)
        ssc.checkpoint('/tmp/checkpoint_v03')
```

Figura 87. Crear contexto de Spark Streaming y directorio de checkpoint

Se ha configurado el contexto de Spark Streaming para que cada 60 segundos se mire la cola Kafka y se realicen las operaciones que se especificarán más adelante. Por otro lado, también hemos definido un directorio de checkpoint (*punto de guardado*), ya que hay una serie de operaciones que lo necesitarán, usando la función **`checkpoint`**. Aquí se irán almacenando los RDD de cada lote para poder utilizarlos en lotes posteriores. Haremos esto porque Spark Streaming funciona de modo que cada lote se procesa de manera independiente, como si los demás lotes no existieran. Esto limita en cierto modo las operaciones posibles, pero lo solventamos con este directorio de checkpoint.

Una vez tenemos el contexto creado, podemos inicializar el streaming a partir de Kafka. Para ello, nos conectaremos al servidor de Zookeper deseado, y leeremos el topic 'twitter'. En el otro lado se encuentra una aplicación consumiendo tweets en streaming, que se van introduciendo en la cola Kafka para ser ingeridos por Spark.

```
In [4]: zookeper = 'tfmkafka.ddns.net:2181'
        kafkaStream = KafkaUtils.createStream(ssc, zookeper, 'spark-streaming',topics={'twitter':1})
```

Figura 88. Streaming Kafka en la aplicación de Spark Streaming

Hemos usado la función **`createStream`** de la librería **`KafkaUtils`** que importamos previamente, pasándole el contexto de Spark Streaming, la dirección del servidor ZooKeeper, el nombre de la aplicación y los topics de Kafka que queremos recibir.

Del flujo recibido, nos quedamos con la segunda componente, que es la que contiene los datos de Kafka. Extraemos el atributo `created_at` por un lado, y el `screen_name` por otro. Para ello usaremos la función **`map`**.

```
In [5]: tweet_dates = kafkaStream.map(lambda x: x[1].split("|")[0])
        tweet_users = kafkaStream.map(lambda x: x[1].split("|")[1])
```

Figura 89. Extracción de los atributos del flujo Kafka

Las siguientes líneas son las que se ejecutarán durante el streaming de Spark. En primer lugar se imprime un extracto de los datos recibidos por cada lote, y a continuación se escribe en un fichero de texto cada una de las fechas de cada RDD del flujo. La parte final se refiere al cálculo de los usuarios más activos del streaming.

```
In [7]: # Count tweets in this batch
        ntweets = kafkaStream.count()
        ntweets.map(lambda x: "Number of tweets this batch: %s" % x).pprint()

        # Print all dates in this batch
        tweet_dates.pprint()
        # Save tweet dates to a file to process it later
        tweet_dates.foreachRDD(lambda x: x.foreach(lambda y: write_dates(y)))

        # Print all users in this batch
        tweet_users.pprint()

        import matplotlib.pyplot as plt

        def updateFunc(newValues, lastSum):
            if lastSum is None:
                lastSum = 0
            result = sum(newValues, lastSum)

            return result

        # Let's count tweet by each user in all the streaming using updateStateByKey function
        count = tweet_users.countByValue().updateStateByKey(updateFunc)
        # Now we sort this users to get the top five
        sortedCount = count.transform(lambda rdd: rdd.sortBy(lambda x: -x[1]))
        sortedCount.pprint(5)


        # This function is used to print a pie chart showing the 5 most active users
        def activeUsers (rdd):
            # Convert RDD to dictionary
            diction = rdd.collectAsMap()
            # Sort list by number of tweets
            sortedList = sorted(diction.items(), key=lambda x: x[1], reverse=True)
            # Create pie chart to show the 5 most active users
            if (len(sortedList) >= 5):
                labels = sortedList[0][0], sortedList[1][0], sortedList[2][0], sortedList[3][0], sortedList[4][0]
                values = [sortedList[0][1], sortedList[1][1], sortedList[2][1], sortedList[3][1], sortedList[4][1]]
                colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue', "#96624E"]
                explode = (0.1, 0, 0, 0, 0)  # explode 1st slice

                plt.pie(values, explode = explode, labels = labels, colors = colors, autopct='%1.1f%%', shadow=True, startangle
                plt.axis('equal')
                plt.show()

        # Now we call this function for each RDD in the DStream
        sortedCount.foreachRDD(lambda x: activeUsers(x))
```

Figura 90. Operaciones a realizar durante el streaming

Lo primero que hacemos es imprimir una serie de datos de cada lote: el número de tweets en el lote, usando **`count`** para realizar el conteo, y **`map`** y **`pprint`** para imprimirlo; una muestra de las fechas recogidas en este lote, con **`pprint`**; y una muestra de los usuarios, también con **`pprint`**.

A continuación se guardan las fechas en un fichero de texto, ya que su procesado se realizará después del streaming, por simplicidad. Usamos la función **foreachRDD**, especificando la función que queremos que se ejecute para cada RDD del lote. Esta función se ha definido aparte, como podemos ver en la siguiente imagen.

```
In [6]: def write_dates (line):
            f = open('dates.txt', 'w')
            f.write(line + '\n')
```

Figura 91. Función para añadir fechas a fichero de texto

Lo último que nos queda por hacer en el streaming es calcular qué usuarios son los más activos. Podemos calcular los tweets de cada usuario utilizando la función **countByValue**. Sin embargo, como hemos dicho, cada lote es independiente de los anteriores, por lo que en principio no sería posible calcularlo en el periodo completo. Para solucionar este problema, Spark permite usar lo que denomina funciones de actualización, siempre y cuando hayamos definido un directorio de checkpoint. Para utilizar las funciones de actualización usamos **updateStateByKey**, especificando la función a ejecutar. En esta función se realiza la suma de los valores anteriores con los recogidos en este nuevo lote.

El siguiente paso es ordenar los valores de manera descendente. Tendremos que ejecutar una función de ordenado sobre cada RDD. Esto se hace usando la función **transform**. Se ejecutará la función **sortBy** sobre los RDD. Una vez los tenemos ordenados, imprimimos los 5 mayores valores, que se corresponderá con los usuarios más activos, utilizando **pprint**.

Finalmente, para obtener una visualización más cómoda de los resultados, usaremos gráficos de tarta, con los usuarios más activos. Usaremos **foreachRDD** para ejecutar una función sobre cada RDD del lote. La función consistirá en:

1) Convertir el RDD en un diccionario, con **collectAsMap**.
2) Crear una lista ordenada, usando **sorted** sobre los elementos del diccionario.
3) Generamos el gráfico de tarta, con la función **pie** de `matplotlib`, a la que habrá que pasarle los valores y etiquetas de los datos, así como otras variables de visualización como los colores de la gráfica.

Finalmente, comenzamos a recibir datos y procesarlos, y esperamos a que termine el procesamiento.

```
In [8]: ssc.start()
        ssc.awaitTermination()
```

Figura 92. Iniciar streaming y esperar su finalización

Cada 60 segundos veremos una salida como la de la siguiente imagen.

```
-------------------------------------------
Time: 2017-04-30 11:46:00
-------------------------------------------
Number of tweets this batch: 11

-------------------------------------------
Time: 2017-04-30 11:46:00
-------------------------------------------
2017-04-30 11:44:52
2017-04-30 11:44:52
2017-04-30 11:44:52
2017-04-30 11:44:52
2017-04-30 11:44:54
2017-04-30 11:44:54
2017-04-30 11:44:54
2017-04-30 11:44:54
2017-04-30 11:44:54
2017-04-30 11:44:55
...

-------------------------------------------
Time: 2017-04-30 11:46:00
-------------------------------------------
StephLuvsSports
LucianoYoma
tweety_re
Tatsuyachan3z
tweety_re
candleclub1
autohebdo
RUBENM98
StSchererZhou
Gerardos92
...

-------------------------------------------
Time: 2017-04-30 11:46:00
-------------------------------------------
('tweety_re', 2)
('RUBENM98', 1)
('StephLuvsSports', 1)
('StSchererZhou', 1)
('autohebdo', 1)
...
```

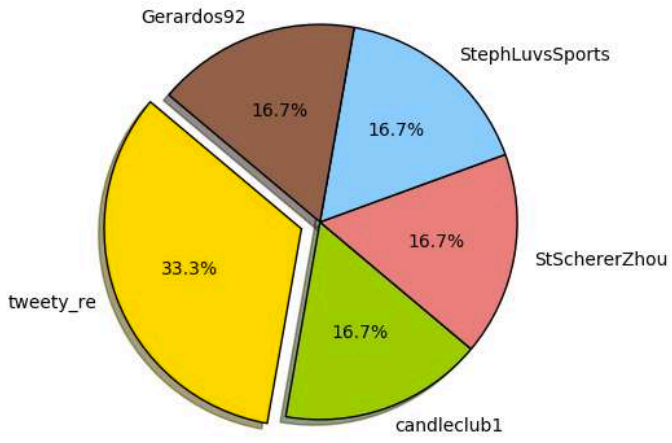

Figura 93. Salida del streaming

Esto se va actualizando cada minuto con los valores del nuevo lote, consiguiendo así realizar el análisis del streaming, en cuanto a los usuarios más activos en este intervalo de tiempo, viendo la variación minuto a minuto.

Al final del streaming, estos son los cinco usuarios más activos.

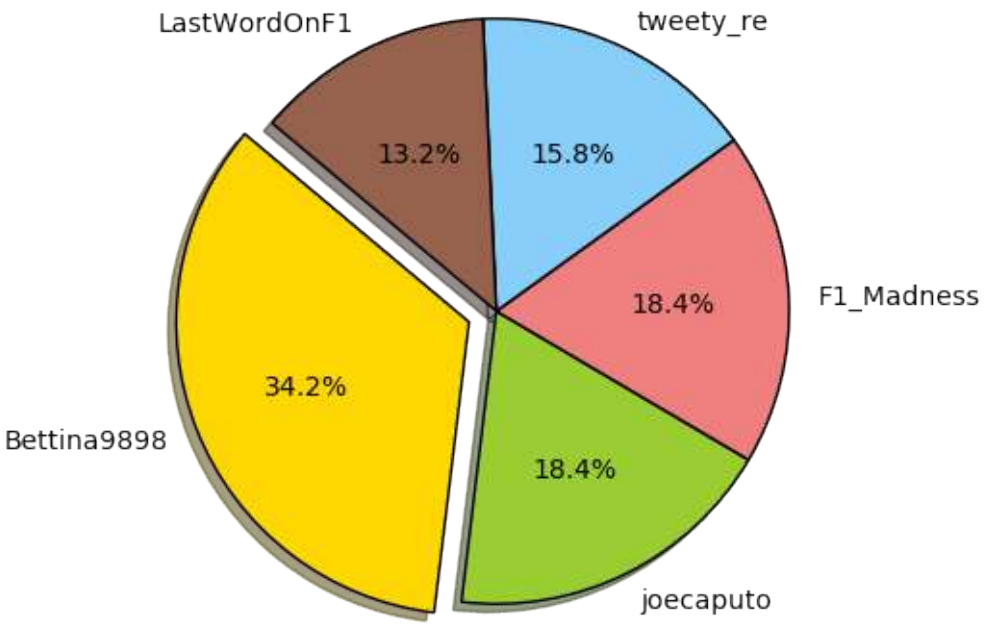

Figura 94. Usuarios más activos del streaming

El procesamiento se puede detener manualmente con la siguiente línea.

```
In [9]: ssc.stop()
```

Figura 95. Detención manual del streaming

Aún nos queda realizar el análisis de frecuencia de tweets. Una vez tenemos todos los datos de las fechas en el fichero de texto, vamos a crear una lista que los contenga todos, para realizar el análisis usando las librerías **`pandas`** y **`numpy`**.

```
In [10]: datefile = "dates.txt"
         total_tweets = 0
         with open(datefile, 'r') as f:
             f.seek(0)
             all_dates = []
             for line in f:
                 all_dates.append(line)
                 total_tweets = total_tweets + 1
```

Figura 96. Crear lista con las fechas del streaming

Simplemente abrimos el fichero y vamos añadiendo las fechas a una lista. También aprovechamos para contar el número total de tweets del streaming.

```
In [11]: print("Total tweets: {}".format(total_tweets))
         Total tweets: 55651
```

Figura 97. Número total de tweets del streaming

De nuevo, añadimos una serie de `imports` para crear una gráfica a partir de los datos obtenidos.

```
In [12]: import pandas as pd
         import numpy as np
         from datetime import datetime
         import matplotlib.pyplot as plt
         import matplotlib.dates as mdates
```

Figura 98. Librerías para hacer gráficas

El siguiente trozo de código utiliza estas librerías para crear una gráfica en la que se mostrará la frecuencia de tweets durante el intervalo de tiempo analizado usando Spark Streaming.

```
In [13]: idx = pd.DatetimeIndex(all_dates)
         ones = np.ones(len(all_dates))
         my_series = pd.Series(ones, index=idx)

         # Resampling / bucketing into 1-minute buckets
         per_minute = my_series.resample('1Min').sum().fillna(0)

         # Plotting the series
         fig, ax = plt.subplots()
         ax.grid(True)
         ax.set_title("Tweet Frequencies")

         hours = mdates.MinuteLocator(interval=20)
         date_formatter = mdates.DateFormatter('%H:%M')

         datemin = datetime(2017, 4, 30, 11, 30)
         datemax = datetime(2017, 4, 30, 13, 50)

         ax.xaxis.set_major_locator(hours)
         ax.xaxis.set_major_formatter(date_formatter)
         ax.set_xlim(datemin, datemax)
         max_freq = per_minute.max()
         ax.set_ylim(0, max_freq)
         ax.plot(per_minute.index, per_minute)

         plt.savefig('tweet_time_series.png')
         plt.show()
```

Figura 99. Código para crear gráfica de frecuencia de tweets

Al ejecutar este código, obtenemos la gráfica de frecuencias de tweets, donde podemos ver la variación temporal de las frecuencias durante todo el intervalo.

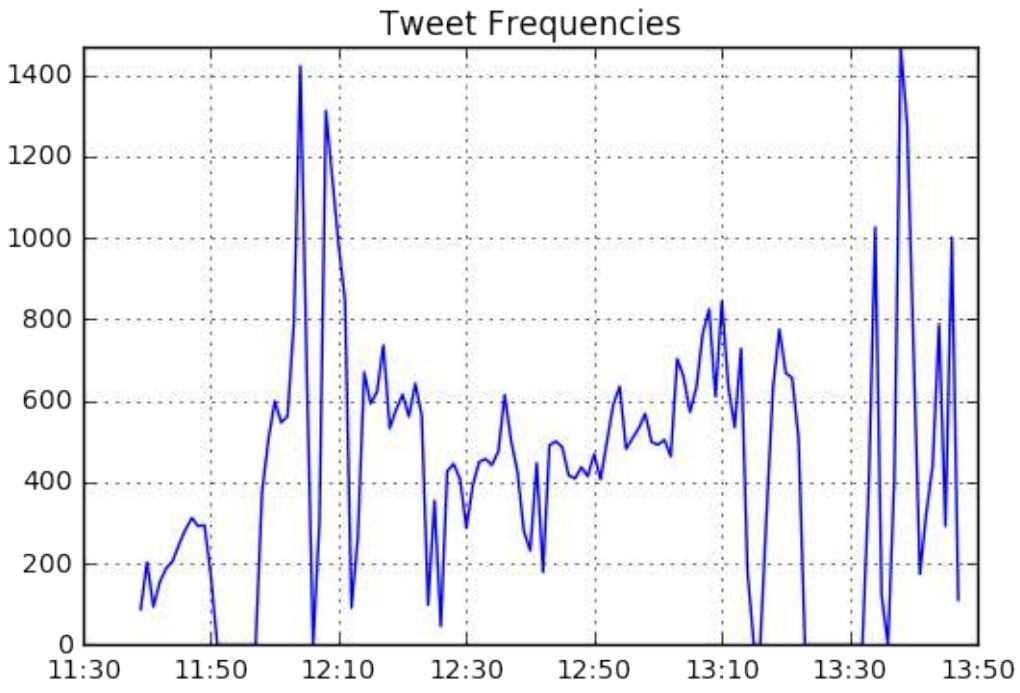

Figura 100. Frecuencias de tweets durante el streaming

La carrera de Fórmula 1 empezó a las 12:00 y terminó a las 13:40. Podemos ver que en estos puntos es donde se alcanzan los picos de la gráfica, con valores por encima de 1400 tweets. El siguiente valor más alto (~1300 tweets) se encuentra en el tiempo 12:08, aproximadamente, que se corresponde con el abandono del piloto español Fernando Alonso de la carrera. Existe una gran variación en la frecuencia de los tweets durante todo el intervalo de la carrera, desde minutos en los que no se publicó ningún tweet sobre este tema, hasta los picos con valores por encima de 1400 tweets.

7 Conclusiones

El trabajo realizado ha abarcado gran parte del ámbito del Big Data, tal y como lo conocemos actualmente. Podemos diferenciar dos grandes piezas que encajan a la perfección y se complementan para conseguir los objetivos propuestos inicialmente.

La primera pieza es la construcción de una plataforma de Big Data, para la cual se han utilizado tecnologías que están a la orden del día, como Apache Spark, Jupyter o Apache Kafka, incluso añadiendo mejoras técnicas con HDFS, Docker y Ansible, para aumentar la escalabilidad y funcionalidad de la plataforma. Además, esta plataforma se ha implementado por completo en la nube de Amazon Web Services, haciendo patente la más que clara y evidente relación entre el Big Data y el Cloud Computing, como tecnologías emergentes que seguirán evolucionando en los próximos años, hasta formar parte, aún más, de nuestro día a día.

Por otro lado, la segunda pieza ha consistido en el procesamiento de datos provenientes de la mayor fuente de datos que existe actualmente para el Big Data: las redes sociales. Aprovechando la completa plataforma de procesamiento que implementamos, se ha podido realizar un procesamiento de ingentes cantidades de datos usando, principalmente, las tecnologías Apache Spark y Apache Kafka, y el lenguaje de programación Python. De hecho, y con el fin de ahondar mucho más en esta potente herramienta, se han diseñado tres aplicaciones que exprimen la funcionalidad de Spark junto con sus módulos Spark SQL y Spark Streaming, que añaden una mayor funcionalidad al procesamiento.

7.1 Líneas de continuación y mejora

Aunque se considera que el trabajo, en su conjunto, es bastante completo e incluye gran variedad de tecnologías, siempre hay posibles mejoras. A continuación se intentará exponer algunas líneas de mejora para nuestro proyecto.

Apache Spark dispone de otros dos módulos, que podrían integrarse perfectamente en el escenario desarrollado. **GraphX** es un módulo de Spark para el procesamiento de grafos, lo cual concuerda a la perfección con la temática de las redes sociales. De hecho, no se ha incluido en este proyecto por no disponer de una API para el lenguaje de programación Python. Por tanto, es una posible línea de continuación si utiliza otro lenguaje como Scala. El otro módulo de Spark es **MLib**, una librería de machine learning para Spark. Permite integrar algoritmos de aprendizaje automático con los procesamientos realizados. No se incluye en este proyecto por considerarse demasiado complejo, considerando el trabajo suficientemente complejo ya. Por este motivo, podría centrarse otro trabajo en aplicar este módulo a un procesamiento de las redes sociales,

por ejemplo, para segmentar a los miembros de la red social en cuanto a algunos atributos de interés para marketing o banca.

En cuanto al procesamiento de datos realizado, podría realizarse un análisis de otros atributos que se pueden extraer de los tweets, como la **geolocalización**. Por ejemplo, una posible aplicación de este aspecto podría ser dibujar en un mapa la localización geográfica de un conjunto de individuos que se encuentre publicando tweets sobre un evento determinado, como en el caso de la carrera de Fórmula 1 de nuestra aplicación de streaming. Esto se puede hacer usando atributos propios de los tweets, como `coordinates`. Se podría obtener una idea general acerca de la localización de la mayoría de los usuarios que estén hablando sobre un tema concreto.

Estas son sólo algunas de las posibles líneas de continuación y mejora de este trabajo, que se quedaron en el tintero a la hora de llevarlo a cabo.

Bibliografía

[1] Ricardo Barranco Fragoso, IBM, "¿Qué es Big Data?" *IBM developer Works*, 2012.

[2] Dennis Hung, "The impact of Big Data in social media marketing strategies" *TECH.CO*, 2016.

[3] Twitter Usage Statistics: http://www.internetlivestats.com/twitter-statistics/, *internet live stats.*

[4] Zephoria, "The top 20 valuable Facebook statistics" *Zephoria posts*, 2017.

[5] Tom Pick, "35 stupendous social networking facts and stats" *Meltwater,* 2016.

[6] Lisa Lowe, "125 amazing social media statistics you should know in 2016" *SocialPilot*, 2016.

[7] Kit Smith, "Marketing: 47 Facebook statistics for 2016" *BrandWatch*, 2016.

[8] Documentación oficial de Apache Spark: http://spark.apache.org/docs/latest/

[9] Documentación oficial de Jupyter: https://jupyter.readthedocs.io/en/latest/

[10] Documentación oficial de Anaconda: https://docs.continuum.io/anaconda/

[11] Documentación oficial de AWS: https://aws.amazon.com/es/documentation/

[12] Documentación oficial de Apache Kafka: https://kafka.apache.org/

[13] Documentación oficial de ZooKeeper: https://zookeeper.apache.org/

[14] Documentación oficial API Twitter: https://dev.twitter.com/overview/api

[15] Cayetano Rodríguez Medina, "Uso de la nube para aplicación distribuida de entrenamiento. Comparativa de distintas plataformas." *Trabajo de Fin de Grado (Universidad de Sevilla)*, 2016.

[16] José Marcial Portilla, "Getting Spark, Python, and Jupyter Notebook running on Amazon EC2" *Medium*, 2016.

[17] Alberto Bonsanto, "Link Spark with iPython Notebook" *Stack Overflow*, 2015.

[18] Alejandro Rodríguez Calzado, Cayetano Rodríguez Medina, "Implementación de un recomendador usando Spark-Scala" *Práctica Big Data (Universidad Politécnica de Madrid)*, 2017.

[19] Documentación oficial Hadoop: https://hadoop.apache.org/docs/r2.7.3/

[20] Documentación oficial Docker: https://docs.docker.com/

[21] Documentación oficial Ansible: http://docs.ansible.com/

[22] Documentación oficial No-IP: https://www.noip.com/

[23] Matei Zaharia, Patrick Wendell, Andy Konwinski, Holden Karau, "Learning Spark" *O'Reilly Media*, ISBN: 978-1-449-35903-4, 2015.

[24] Matthew A. Russell, "Mining the Social Web" *O'Reilly Media*, ISBN: 978-1-449-38834-8, 2011.

[25] Marco Bonzanini, "Mastering Social Media Mining with Python" *Packt Publishing*, ISBN: 978-1-78355-201-6, 2016.

[26] Documentación oficial de Tweepy: http://docs.tweepy.org/en/v3.5.0/

[27] Documentación oficial API WordCloud: https://amueller.github.io/word_cloud/

[28] Robin Moffatt, "Getting Started with Spark Streaming, Python, and Kafka" *rittmanmead*, 2017.

Anexos

A modo de anexo se adjunta el código de las aplicaciones desarrolladas a lo largo del trabajo. También es posible encontrar el código en **GitHub**: https://github.com/cayrodmed1/tfm/.

twitter_get_user.py

```
import os
import sys
import json
import time
import math
from tweepy import Cursor, API, OAuthHandler

MAX_FRIENDS = 15000

def usage():
  print("Usage:")
  print("python {} <username>".format(sys.argv[0]))

def paginate(items, n):
  """Generate n-sized chunks from items"""
  for i in range(0, len(items), n):
    yield items[i:i+n]

if __name__ == '__main__':
  if len(sys.argv) != 2:
    usage()
    sys.exit(1)
  screen_name = sys.argv[1]

  ACCESS_TOKEN = 'XXXXXXXXXXXXXXXXXXXXXXX'
  ACCESS_SECRET = 'XXXXXXXXXXXXXXXXXXXXXXXXX'
  CONSUMER_KEY = 'XXXXXXXXXXXXXXXXXXXXX'
  CONSUMER_SECRET = 'XXXXXXXXXXXXXXXXXXX'

  auth = OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)
  auth.set_access_token(ACCESS_TOKEN, ACCESS_SECRET)
  client = API(auth)
  dirname = "users/{}".format(screen_name)
  max_pages = math.ceil(MAX_FRIENDS / 5000)
  try:
    os.makedirs(dirname, mode=0o755, exist_ok=True)
```

```
  except OSError:
    print("Directory {} already exists".format(dirname))
  except Exception as e:
    print("Error while creating directory {}".format(dirname))
    print(e)
    sys.exit(1)

  # get followers for a given user
  fname = "users/{}/followers.jsonl".format(screen_name)
  with open(fname, 'w') as f:
    for followers in Cursor(client.followers_ids,
                            screen_name=screen_name).pages(max_pages):
      for chunk in paginate(followers, 100):
        users = client.lookup_users(user_ids=chunk)
        for user in users:
          f.write(json.dumps(user._json)+"\n")
      if len(followers) == 5000:
        print("More results available. Sleeping for 60 seconds to  avoid rate
limit")
        time.sleep(60)

  # get friends for a given user
  fname = "users/{}/friends.jsonl".format(screen_name)
  with open(fname, 'w') as f:
    for friends in Cursor(client.friends_ids,
                          screen_name=screen_name).pages(max_pages):
      for chunk in paginate(friends, 100):
        users = client.lookup_users(user_ids=chunk)
        for user in users:
          f.write(json.dumps(user._json)+"\n")
      if len(friends) == 5000:
        print("More results available. Sleeping for 60 seconds to avoid rate
limit")
        time.sleep(60)

  # get user's profile
  fname = "users/{}/user_profile.json".format(screen_name)
  with open(fname, 'w') as f:
    profile = client.get_user(screen_name=screen_name)
    f.write(json.dumps(profile._json, indent=4))
```

twitter_get_user_timeline.py

```
import sys
import json
from tweepy import Cursor, API, OAuthHandler

if __name__ == '__main__':
  user = sys.argv[1]

  ACCESS_TOKEN = 'XXXXXXXXXXXXXXXXXXXXXXXX'
  ACCESS_SECRET = 'XXXXXXXXXXXXXXXXXXXXXXXXX'
  CONSUMER_KEY = 'XXXXXXXXXXXXXXXXXXXXX'
  CONSUMER_SECRET = 'XXXXXXXXXXXXXXXXXXXXX'

  auth = OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)
  auth.set_access_token(ACCESS_TOKEN, ACCESS_SECRET)
  client = API(auth)

  fname = "user_timeline_{}.jsonl".format(user)

  with open(fname, 'w') as f:
    for page in Cursor(client.user_timeline, screen_name=user,
                       count=200).pages(16):
      for status in page:
        if (status._json['text'].split(' ')[0] != 'RT'):
          f.write(json.dumps(status._json)+"\n")
```

twitter-kafka.py

```
import json
import tweepy
import string

from pykafka import KafkaClient

# Twitter API tokens
ACCESS_TOKEN = 'XXXXXXXXXXXXXXXXXX'
ACCESS_SECRET = 'XXXXXXXXXXXXXXXXXXX'
CONSUMER_KEY = 'XXXXXXXXXXXXXXXXXXXXX'
CONSUMER_SECRET = 'XXXXXXXXXXXXXXXXXXXX'

# Listener class to be used by tweepy
class StdOutListener(tweepy.StreamListener):
```

```
    # When a tweet comes, send attributes 'created_at' and 'screen_name' to
Kafka
    def on_status(self, status):
        date = str(status.created_at)
        user = status._json['user']['screen_name']
        msg = date + "|" + user
        with mytopic.get_sync_producer() as producer:
            producer.produce(str(msg))
        return True


    def on_error(self, status_code):
        print('Got an error with status code: ' + str(status_code))
        return True

    def on_timeout(self):
        print('Timeout...')
        return True

# Main function
if __name__ == '__main__':

    listener = StdOutListener()

    oauth = tweepy.OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)
    oauth.set_access_token(ACCESS_TOKEN, ACCESS_SECRET)

    # Create twitter stream using OAuth credentials and the listener
    stream = tweepy.Stream(oauth, listener)

    # We need to specify the Kafka server
    client = KafkaClient(hosts="tfmkafka.ddns.net:9092")

    # This is the topic in the Kafka broker
    mytopic = client.topics['twitter']

    # Here we can filter tweets by any word
    stream.filter(track=['#RussianGP'])
```

user_analysis.ipynb

```
from pyspark import SparkContext
from pyspark import SparkConf
conf = SparkConf().setAppName("user")
                  .setMaster("spark://tfmspark.ddns.net:7077")
sc = SparkContext.getOrCreate(conf=conf)


screen_name = 'CayeRodriguez'

import json
hdfs = 'hdfs://tfmhdfs.ddns.net:9000/user/centos/files/'
followers_file = 'users/{}/followers.jsonl'.format(screen_name)
reach = sc.textFile(hdfs + followers_file).map(lambda x:
json.loads(x)['followers_count'])

sum_reach = reach.sum()
nfollowers = reach.count()
avg_followers = round(sum_reach / nfollowers, 2)

friends_file = 'users/{}/friends.jsonl'.format(screen_name)
followers = sc.textFile(hdfs + followers_file).map(lambda x:
json.loads(x)['screen_name'])
friends = sc.textFile(hdfs + friends_file).map(lambda x:
json.loads(x)['screen_name'])
nfriends = friends.count()
mutual = friends.intersection(followers).count()
friends_not_following = nfriends - mutual
followers_not_followed = nfollowers - mutual

print("{} followers".format(nfollowers))
print("{} friends".format(nfriends))
print("{} mutual friends".format(mutual))
print("{} friends are not following back"
      .format(friends_not_following))
print("{} followers are not followed back by {}"
      .format(followers_not_followed, screen_name))
print("{} users reached by 1-degree connections".format(sum_reach))
print("Average number of followers for {}'s followers: {}"
      .format(screen_name, avg_followers))

timeline_file = 'user_timeline_{}.jsonl'.format(screen_name)
timeline = sc.textFile(hdfs + timeline_file)
favorite_count = timeline.map(lambda x:
                              json.loads(x)['favorite_count']).sum()
retweet_count = timeline.map(lambda x:
                              json.loads(x)['retweet_count']).sum()
ntweets = timeline.count()
avg_favorite = round(favorite_count / ntweets, 2)
avg_retweet = round(retweet_count / ntweets, 2)
favorite_per_user = round(favorite_count / nfollowers, 2)
retweet_per_user = round(retweet_count / nfollowers, 2)
```

```
print("Total number of tweets: {}".format(ntweets))
print("Favorited {} times ({} per tweet, {} per user)"
      .format(favorite_count, avg_favorite, favorite_per_user))
print("Retweeted {} times ({} per tweet, {} per user)"
      .format(retweet_count, avg_retweet, retweet_per_user))


texts = timeline.map(lambda x: json.loads(x)['text'])
texts.collect()

import string
import re
clean = texts.map(lambda x: x.lower())
           .map(lambda x: re.sub(r"http\S+", "", x))
           .map(lambda x: re.sub(r"@\S+", "", x))
           .map(lambda x: re.sub(r"\\\S+", "", x))
           .map(lambda x: "".join(c for c in x if c not in
                ('!','.', ',',':', ';', '?', '(', ')', '[', ']')))
           .map(lambda x: ''.join(i for i in x if not i.isdigit()))
clean.collect()


words = clean.flatMap(lambda x: x.split(' '))
words.collect()

wordPairs = words.map(lambda x: (x,1))
wordPairs.collect()

wordCounts = wordPairs.reduceByKey(lambda x, y: x+ y)
wordCounts.collect()

wordCountsDict = wordCounts.collectAsMap()
del (wordCountsDict[''])
del (wordCountsDict['"'])
del (wordCountsDict['-'])
delete_list = ["rt", "que", "de", "a", "y", "no", "si", "el", "la",
"los", "las", "un", "una", "unos", "unas", "en", "lo", "es", "ya",
"por", "yo", "tu", "me", "se", "te", "con", "mi", "para", "he",
"ha", "o", "al", "pa", "hay", "del", "mas", "más", "como", "has",
"ni", "sin", "eso", "le", "este", "esta", "hace", "hacer", "pero",
"hasta", "va", "porque", "pues", "menos", "han", "to", "tiene",
"asi", "así", "aquí", "algo", "x", "ir", "son", "estoy", "voy",
"nos", "muy", "mucho", "luego", "estos", "estas"]
for word in delete_list:
      del (wordCountsDict[word])

sortedList = sorted(wordCountsDict.items(), key=lambda x: x[1],
                    reverse=True)
sortedList


from wordcloud import WordCloud, STOPWORDS


# Generate a word cloud image
wordcloud = WordCloud(max_font_size=40)
               .generate_from_frequencies(sortedList)
```

```
import matplotlib.pyplot as plt
plt.figure()
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()
```

trends.ipynb

```
from pyspark import SparkContext
from pyspark import SparkConf
conf = SparkConf().setAppName("trends")
                  .setMaster("spark://tfmspark.ddns.net:7077")
sc = SparkContext.getOrCreate(conf=conf)

import os
import sys
from tweepy import API
from tweepy import OAuthHandler

# Variables that contains the user credentials to access Twitter
API
ACCESS_TOKEN = 'XXXXXXXXXXXXXXXX'
ACCESS_SECRET = 'XXXXXXXXXXXXXXXX'
CONSUMER_KEY = 'XXXXXXXXXXXXXX'
CONSUMER_SECRET = 'XXXXXXXXXXXXXXXXX'

auth = OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)
auth.set_access_token(ACCESS_TOKEN, ACCESS_SECRET)
```

```
client = API(auth)


trends = sc.parallelize(client.trends_available())

trends.collect()


from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Spark SQL trends")

            .getOrCreate()


df = spark.read.json(trends)

df.show()

df.printSchema()


df.select("name", "woeid").filter(df['country']=='Spain').show()


spain_trends = client.trends_place(23424950)


import json

with open('spain_trends.json', 'w') as outfile:

     json.dump(spain_trends[0]['trends'], outfile)


df_spain = spark.read.json('spain_trends.json')

df_spain.show()

df_spain.printSchema()


df_spain.select('name').show(10)


world_trends = client.trends_place(1)


with open('world_trends.json', 'w') as outfile:

       json.dump(world_trends[0]['trends'], outfile)
```

```
df_world = spark.read.json('world_trends.json')
df_world.select('name').show(10)
```

streaming.ipynb

```
from pyspark import SparkContext
from pyspark import SparkConf
conf = SparkConf().setAppName("streaming")
                  .setMaster("spark://tfmspark.ddns.net:7077")
sc = SparkContext.getOrCreate(conf=conf)


# Spark Streaming
from pyspark.streaming import StreamingContext
# Kafka
from pyspark.streaming.kafka import KafkaUtils

ssc = StreamingContext(sc, 60)
ssc.checkpoint('/tmp/checkpoint_v03')


zookeper = 'tfmkafka.ddns.net:2181'
kafkaStream = KafkaUtils.createStream(ssc, zookeper,
                        'spark-streaming',topics={'twitter':1})


tweet_dates = kafkaStream.map(lambda x: x[1].split("|")[0])
tweet_users = kafkaStream.map(lambda x: x[1].split("|")[1])


def write_dates (line):
      f = open('dates.txt', 'w')
      f.write(line + '\n')
```

```
# Count tweets in this batch
ntweets = kafkaStream.count()
ntweets.map(lambda x: "Number of tweets this batch: %s"
                        % x).pprint()
# Print all dates in this batch
tweet_dates.pprint()
# Save tweet dates to a file to process it later
tweet_dates.foreachRDD(lambda x:
                        x.foreach(lambda y: write_dates(y)))
# Print all users in this batch
tweet_users.pprint()


import matplotlib.pyplot as plt
def updateFunc(newValues, lastSum):
      if lastSum is None:
            lastSum = 0
      result = sum(newValues, lastSum)
      return result


# Let's count tweet by each user in all the streaming using
updateStateByKey function
count = tweet_users.countByValue().updateStateByKey(updateFunc)
# Now we sort this users to get the top five
sortedCount = count.transform(lambda rdd:
                  rdd.sortBy(lambda x: -x[1])) sortedCount.pprint(5)


# This function is used to print a pie chart showing the 5 most
active users
def activeUsers (rdd):
```

```
        # Convert RDD to dictionary

        diction = rdd.collectAsMap()

        # Sort list by number of tweets

        sortedList = sorted(diction.items(), key=lambda x: x[1],
reverse=True)

        # Create pie chart to show the 5 most active users

        if (len(sortedList) >= 5):

            labels = sortedList[0][0], sortedList[1][0],
        sortedList[2][0], sortedList[3][0], sortedList[4][0]

            values = [sortedList[0][1], sortedList[1][1],
        sortedList[2][1], sortedList[3][1], sortedList[4][1]]

            colors = ['gold', 'yellowgreen', 'lightcoral',
        'lightskyblue', "#96624E"]

            explode = (0.1, 0, 0, 0, 0)  # explode 1st slice

            plt.pie(values, explode = explode, labels = labels,
        colors = colors, autopct='%1.1f%%', shadow=True,
        startangle=140)

            plt.axis('equal')

            plt.show()


# Now we call this function for each RDD in the DStream

sortedCount.foreachRDD(lambda x: activeUsers(x))



ssc.start()

ssc.awaitTermination()

ssc.stop()


datefile = "dates.txt"

total_tweets = 0
```

```
with open(datefile, 'r') as f:
      f.seek(0)
      all_dates = []
      for line in f:
            all_dates.append(line)
            total_tweets = total_tweets + 1

print("Total tweets: {}".format(total_tweets))

import pandas as pd
import numpy as np
from datetime import datetime
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

idx = pd.DatetimeIndex(all_dates)
ones = np.ones(len(all_dates))
my_series = pd.Series(ones, index=idx)
# Resampling / bucketing into 1-minute buckets
per_minute = my_series.resample('1Min').sum().fillna(0)
# Plotting the series
fig, ax = plt.subplots()
ax.grid(True)
ax.set_title("Tweet Frequencies")
hours = mdates.MinuteLocator(interval=20)
date_formatter = mdates.DateFormatter('%H:%M')
datemin = datetime(2017, 4, 30, 11, 30)
datemax = datetime(2017, 4, 30, 13, 50)
```

```
ax.xaxis.set_major_locator(hours)
ax.xaxis.set_major_formatter(date_formatter)

ax.set_xlim(datemin, datemax)

max_freq = per_minute.max()

ax.set_ylim(0, max_freq)

ax.plot(per_minute.index, per_minute)

plt.savefig('tweet_time_series.png')

plt.show()
```