Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

ANÁLISIS DE ARQUITECTURAS DE PROCESADO DE STREAMING BIG DATA

TRABAJO FIN DE MÁSTER

Mario Pérez Esteso

2015

Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

Máster Universitario en Ingeniería de Redes y Servicios Telemáticos

TRABAJO FIN DE MÁSTER

ANÁLISIS DE ARQUITECTURAS DE PROCESADO DE STREAMING BIG DATA

Autor
Mario Pérez Esteso

Director
Juan Carlos Dueñas López

Departamento de Ingeniería de Sistemas Telemáticos

2015

Resumen

En este proyecto se pretende realizar un análisis y exhaustivo estudio de las arquitecturas existentes de procesado de streaming Big Data. Existen múltiples tecnologías con las que se pueden tratar datos masivos y que se pueden utilizar de forma conjunta, de forma que si es necesario para la resolución de un problema unas tecnologías pueden complementar a otras. Debido a esto, el proyecto se ha desarrollado en un entorno multidisciplinar, ya que requiere de conocimientos de Big data, ciencia de datos y machine learning. Tras realizar el análisis de las tecnologías existentes, se procederá a implementar casos prácticos haciendo uso de Apache Spark y programados en diferentes lenguajes: Scala y Python.

Palabras clave: Big-Data, Streaming, Machine Learning, Apache Spark.

Índice general

Índice de figuras

Índice de fragmentos de código

Capítulo 1

Introducción

Vivimos actualmente en un mar de datos que está en constante expansión. Dichos datos son analizados y estudiados para obtener información útil y valiosa sobre ellos. El problema es que en muchos casos estos datos deben ser tratados con tecnologías diseñadas específicamente para este fin.

1.1. Conceptos básicos del dominio

En esta sección se introducirán los conceptos básicos necesarios para comprender el contexto en el que se sitúa el trabajo. Se realizará una explicación de los tipos de analítica existentes, así como de los fundamentos del Big Data y la ciencia de datos.

1.1.1. Analítica predictiva y descriptiva

Existen dos tipos de analítica según el caso de estudio: analítica descriptiva y analítica predictiva. A grandes rasgos, la analítica descriptiva permite saber qué ha sudecido en un tiempo pasado mediante el análisis de los datos. Sin embargo, gracias a la analítica predictiva es posible conocer lo que podría

pasar en un tiempo futuro.

Llevar a cabo el análisis de datos históricos es una técnica muy utilizada por ejemplo en *business intelligence*. A partir de datos obtenidos de hechos que ya han ocurrido es posible obtener distintas estadísticas como por ejemplo la ubicación de los clientes, qué mes es en el que más se vende o qué producto es el más solicitado. Gracias a esto es posible tomar decisiones sólidas basadas en hechos. Sin embargo, aprovechando la información recogida y los patrones obtenidos al analizar los datos es posible predecir qué va a pasar en un futuro. Es ahora cuando la analítica predictiva entra en juego.

La analítica predictiva se basa en el descubrimiento de patrones ocultos en datos que una persona puede no ver, o que al menos le resultaría complicado. Es posible aplicar la analítica predictiva a una gran cantidad de conjuntos de datos en la industria.

1.1.2. Big Data

El Big Data ha revolucionado la manera en la que funciona la industria. Actualmente existen herramientas para recoger, almacenar y procesar información en grandes dimensiones. Tiene cuatro reglas o 'V': volumen, velocidad, variedad, veracidad.

- **Volumen:** según IBM, se generan a diario 2.5 trillones de bytes en Internet, es decir, se generan datos continuamente. Debido a esto, el volumen de datos con el que hay que trabajar y que hay que analizar en enorme.

- **Velocidad:** es extremadamente importante cumplir el requisito de la velocidad. El análisis en tiempo real es vital para la actividad de cualquier empresa. A veces hay que tomar decisiones con un rango de tiem-

po ínfimo, por lo que el flujo de información debe ser rápido.

- **Variedad:** no todos los datos ienen igual estructura. Hay datos que deben cruzarse para ser analizados y obtener información sobre ellos.
- **Veracidad:** en el entorno de Big Data una mayor variedad implica una mayor incertidumbre sobre los datos.

En este proyecto van a desarrollarse algoritmos de machine learning sobre un flujo de datos mediante tecnologías Big Data. En este caso, es importante el uso de Big Data porque se va a manejar una cantidad de datos masiva y es necesario cumplir con alguna de las cuatro reglas anteriormente citadas.

1.1.3. Ciencia de datos

El arte de analizar todos estos datos es denominado ciencia de datos y su objetivo es obtener conocimiento de manera automatizada a partir de un conjunto de datos dado. La ciencia de datos está ayudando a crear nuevas ramas en la ciencia y está influenciando sectores como el de humanidades o ciencias sociales.

Mediante la combinación de estadística, informática, matemáticas y visualización es posible obtener mucha información que no podríamos disponer sin un estudio exhaustivo de los datos.

Es necesario mencionar varios puntos para comprender la metodología de trabajo de un científico de datos. La mayoría del trabajo que se realiza para llevar a cabo análisis de datos eficientes consiste en el preprocesamiento de datos. Normalmente la fuente de la información está desordenada e incompleta y más en grandes conjuntos de datos, en los que hay que utilizar métodos por software para preprocesar los datos. A continuación, tras haber formateado los datos para poder trabajar de forma eficiente, hay que iterar

varias veces sobre ellos para analizarlos y obtener un modelo de predicción. Es realmente complicado conseguirlo a la primera, por lo que hay que analizarlos varias veces. Es un proceso que requiere de mucha experimentación. Aparte de construir modelos, uno de los objetivos de los científicos de datos es poner dichos modelos a disposición de personas ajenas a la ciencia de datos. No sirve de nada tener un programa que obtenga patrones a partir de datos si luego no se lleva a una aplicación en producción.

1.2. Objetivos

Los objetivos de este trabajo han sido divididos en objetivos generales y objetivos específicos.

1.2.1. Objetivo general

Analizar las arquitecturas de procesado de streaming Big Data

Este objetivo consiste en realizar un análisis de las arquitecturas, herramientas y tecnologías existentes que se utilizan para tratar streamings masivos de datos.

1.2.2. Objetivos específicos

Analizar Apache Spark

Este objetivo tiene que ver con el estudio y análisis en profundidad de la herramienta Apache Spark. Para cumplir el resto de objetivos hay que conseguir primero este.

Dado que en este proyecto se van a realizar tres casos de estudio diferentes con procesado de datos en streaming y herramientas de machine learning,

conviene conocer bien el funcionamiento de Apache Spark a nivel interno para una resolución óptima de los casos de estudio propuestos.

Contruir y entrenar modelos

Se crearán y entrenarán modelos de predicción para los casos de estudio propuestos. Dichos modelos dependerán de los algoritmos utilizados y habrá que tener en cuenta el tipo de datos que se reciben, así como su origen y posterior tratamiento con cada herramienta.

Capítulo 2

Tecnologías habilitantes

Existen varias herramientas utilizadas para analizar datos. Durante mucho tiempo, con frameworks open source como R, Octave o Python se han llevado a cabo análisis rápidos y construcciones de modelos con relativamente pequeños conjuntos de datos. Con menos de 10 líneas de código es posible crear un modelo, entrenarlo y ponerlo. Con un poco más de esfuerzo, es posible construir más modelos y compararlos unos con otros para ver cuál es el mejor. Sin embargo, esto se realiza en una sola máquina. Si necesitáramos analizar conjuntos de datos tan grandes que un ordenador personal no pudiera manejarlos habría que recurrir a otro tipo de herramientas.

La solución que se podría pensar sería en ejecutar estos scripts desarrollados en R o Python en múltiples máquinas y configurarlos para ser ejecutados de forma distribuida. Sin embargo, la computación distribuida no es tan sencilla como la centralizada. Los datos deben dividirse y separarse entre varios nodos de un cluster, pero los algoritmos que ejecutan los modelos tienen dependencia de datos entre ellos. Cuantas más máquinas estén trabajando en un problema, más probabilidades hay de que algo falle.

2.1. Sistemas Big Data

En esta sección se realizará un análisis de las principales tecnologías Big Data existentes. Actualmente se utilizan muchas herramientas para tratamiento masivo de datos, pero las más importantes son Apache Hadoop y Apache Spark.

2.1.1. Apache Hadoop

Apache Hadoop es un framework de código abierto que permite el procesamiento distribuido de grandes conjuntos de datos en varios clusters de ordenadores, pero que a ojos del usuario parece un único ordenador. Hadoop separa y distribuye automáticamente los archivos que contienen los datos, además de dividir el trabajo en tareas más pequeñas y ejecutarlas de manera distribuida y recuperarse de posibles fallos automáticamente y de forma transparente al usuario.

El proyecto Hadoop consta de los siguientes módulos:

- **Hadoop Common:** proporciona el acceso a los sistemas de archivos soportados por Hadoop y contiene el código necesario para poder ejecutar el framework. Además, se pone a disposición del usuario el código fuente y la documentación necesaria para aprender a utilizar la herramienta.

- **Hadoop Distributed File System:** HDFS [3] es un sistema de ficheros distribuido altamente tolerante a fallos y diseñado para utilizarse en hardware de bajo coste. Proporciona un alto rendimiento en el acceso a datos y se adapta sin dificultad a aplicaciones con grandes conjuntos de

datos. Fue desarrollado originalmente para el proyecto Apache Nutch, un motor de búsquedas web.

- **Hadoop YARN (Yet Another Resource Negociator):** YARN es una tecnología de gestión de clusters. Se compone de un gestor de recursos central y un gestor por cada nodo, que se ocupa de controlar un único nodo.

- **Hadoop MapReduce:** fue desarrollado para hacer frente a problemas de manera distribuida, los cuales comparten ciertas similitudes pero que requieren desarrollos completos desde el inicio. En datos distribuidos es común tener dos fases: fase de mapeado (*map phase*) y fase de reducción (*reduce phase*). La fase de mapeado trabaja sobre datos sin procesar y produce valores intermedios que pasan a la fase de reducción para producir la salida final de MapReduce. Con esta tecnología se evita el tener que diseñar distintos modelos para resolver problemas similares.

Funcionamiento de MapReduce

El paradigma de MapReduce [4] se basa en enviar el proceso computacional al sitio donde residen los datos que se van a tratar, los cuales se coleccionan en un cluster Hadoop. Cuando se lanza un proceso de MapReduce se distribuyen las tareas entre los diferentes servidores del cluster y, es el propio framework Hadoop quien gestiona el envío y recepción de datos entre nodos. Mucha de la computación sucede en los nodos que tienen los datos en local para minimizar el tráfico de red. Una vez se han procesado todos los datos, el usuario recibe el resultado del cluster. Basicamente Hadoop se ocupa de todo.

Se puede decir que MapReduce es un modelo de programación para computación distribuida basado en Java, pero que también se puede desa-

rrollar en otros lenguajes de programación. Contiene dos fases, aunque la segunda se subdivide en otras dos:

- Map.
- Reduce:
  - Barajado de datos.
  - Reduce.

Si un programa está desarrollado correctamente, puede perfectamente escalar en cuanto a tratamiento de datos de 1MB a 1TB sin ningún problema.

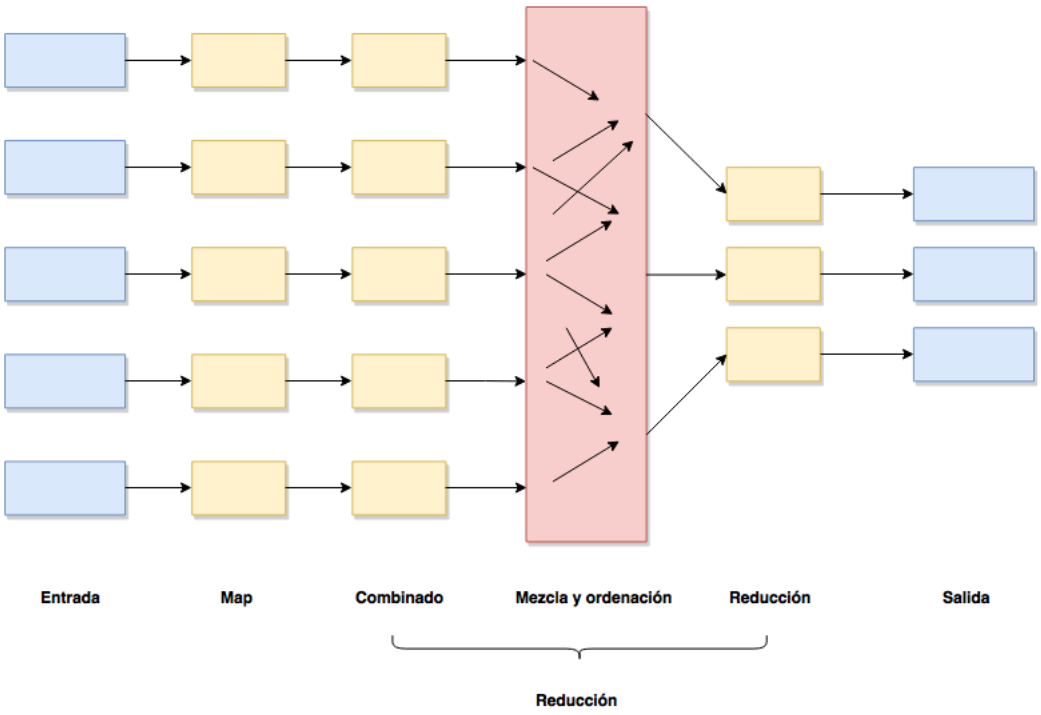


Figura 2.1: Flujo de trabajo de MapReduce.

Un ejemplo de aplicación de MapReduce es obtener las diferentes palabras de un texto y el número de veces que aparecen. Como entrada de datos,

MapReduce recibe el texto y automáticamente lo separa en diferentes fragmentos, tal y como se aprecia en la figura 2.2. Dichos fragmentos podrían ser las diferentes líneas del texto, las cuales se van a mapear para obtener las palabras que forman dicha línea. Puede que se tenga una palabra dos veces, por lo que se unirán en el proceso de combinado. Si hay dos palabras iguales, serán dos en total. A continuación, en el barajado y reordenación de los datos se combinarán los datos de todos los fragmentos, para luego hacer la reducción y devolver un resultado.

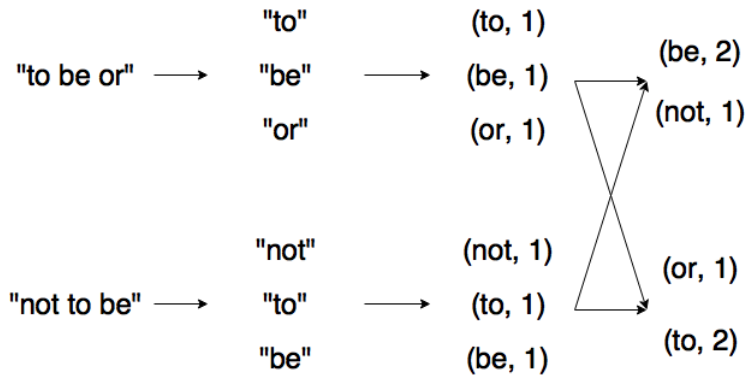


Figura 2.2: Ejemplo de MapReduce para contar palabras.

2.1.2. Apache Spark

Apache Spark combina un sistema de computación distribuida a través de clusters de ordenadores con una manera sencilla y elegante de escribir programas. Fue creado en la Universidad de Berkeley en California y es considerado el primer software de código abierto que hace la programación distribuida realmente accesible a los científicos de datos.

Es sencillo entender Spark si lo comparamos con su predecesor, MapReduce, el cual revolucionó la manera de trabajar con grandes conjuntos de datos ofreciendo un modelo relativamente simple para escribir programas que se podían ejecutar paralelamente en cientos y miles de máquinas al mis-

mo tiempo. Gracias a su arquitectura, MapReduce logra prácticamente una relación lineal de escalabilidad, ya que si los datos crecen es posible añadir más máquinas y tardar lo mismo.

Spark mantiene la escalabilidad lineal y la tolerancia a fallos de MapReduce, pero amplía sus bondades gracias a varias funcionalidades: DAG y RDD.

DAG (Directed Acyclic Graph)

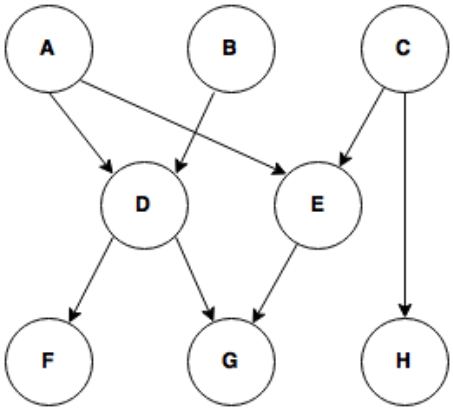


Figura 2.3: Ejemplo de un DAG con 8 nodos.

DAG (Grafo Acíclico Dirigido) es un grafo dirigido que no tiene ciclos, es decir, para cada nodo del grafo no hay un camino directo que comience y finalice en dicho nodo. Un vértice se conecta a otro, pero nunca a si mismo.

Spark soporta el flujo de datos acíclico. Cada tarea de Spark crea un DAG de etapas de trabajo para que se ejecuten en un determinado cluster. En comparación con MapReduce, el cual crea un DAG con dos estados predefinidos (Map y Reduce), los grafos DAG creados por Spark pueden tener cualquier número de etapas. Spark con DAG es más rápido que MapReduce por el hecho de que no tiene que escribir en disco los resultados obtenidos en las etapas intermedias del grafo. MapReduce, sin embargo, debe escribir en disco los resultados entre las etapas Map y Reduce.

Gracias a una completa API, es posible programar complejos hilos de ejecución paralelos en unas pocas líneas de código.

RDD (Resilient Distributed Dataset)

Apache Spark mejora con respecto a los demás sistemas en cuanto a la computación en memoria. RDD [7] permite a los programadores realizar operaciones sobre grandes cantidades de datos en clusters de una manera rápida y tolerante a fallos. Surge debido a que las herramientas existentes tienen problemas que hacen que se manejen los datos ineficientemente a la hora de ejecutar algoritmos iterativos y procesos de minería de datos. En ambos casos, mantener los datos en memoria puede mejorar el rendimiento considerablemente.

Una vez que los datos han sido leídos como objetos RDD en Spark, pueden realizarse diversas operaciones mediante sus APIs. Los dos tipos de operaciones que se pueden realizar son:

- **Transformaciones:** tras aplicar una transformación, obtenemos un nuevo y modificado RDD basado en el original.
- **Acciones:** una acción consiste simplemente en aplicar una operación sobre un RDD y obtener un valor como resultado, que dependerá del tipo de operación.

Dado que las tareas de Spark pueden necesitar realizar diversas acciones o transformaciones sobre un conjunto de datos en particular, es altamente recomendable y beneficioso en cuanto a eficiencia el almacenar RDDs en memoria para un rápido acceso a los mismos. Mediante la función **cache()** se almacenan los datos en memoria para que no sea necesario acceder a ellos en disco.

El almacenamiento de los datos en memoria caché hace que los algoritmos de machine learning ejecutados que realizan varias iteraciones sobre el conjunto de datos de entrenamiento sea más eficiente. Además, se pueden almacenar versiones transformadas de dichos datos.

Modelo de programación

Un programa típico se organiza de la siguiente manera:

1. A partir de una variable de entorno llamada `context` se crea un objeto RDD leyendo datos de fichero, bases de datos o cualquier otra fuente de información.

2. Una vez creado el RDD inicial se realizan transformaciones para crear más objetos RDD a partir del primero. Dichas transformaciones se expresan en términos de programación funcional y no eliminan el RDD original, sino que crean uno nuevo.

3. Tras realizar las acciones y transformaciones necesarias sobre los datos, los objetos RDD deben converger para crear el RDD final. Este RDD puede ser almacenado.

Un pequeño ejemplo de código en Python que cuenta el número de palabras que contiene un archivo sería el siguiente:

```
1 my_RDD = spark.textFile("hdfs://...")
2 words = my_RDD.flatMap(lambda line: line.split(" "))
3               .map(lambda word: (word, 1))
4               .reduceByKey(lambda a, b: a + b)
5 words.saveAsTextFile("hdfs://...")
```

Fragmento de código 2.1: Contar las palabras de un fichero.

Cuando el programa comienza su ejecución crea un grafo similar al de la figura 2.4 en el que los nodos son objetos RDD y las uniones entre ellos son operaciones de transformación. El grafo de la ejecución es un DAG y, cada grafo es una unidad atómica de ejecución. En la figura 2.4, las líneas rojas representan transformación y las verdes operación.

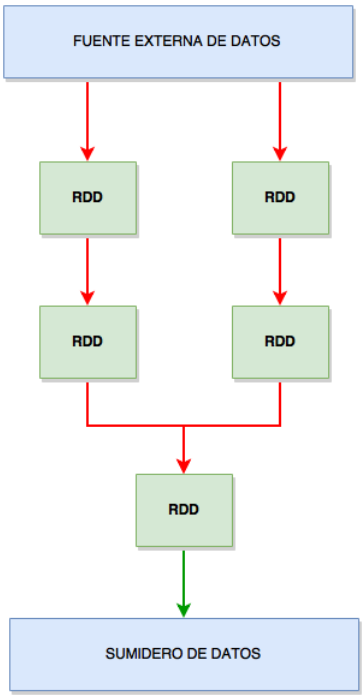


Figura 2.4: Proceso de lectura y transformación de un RDD.

Tipos de transformaciones

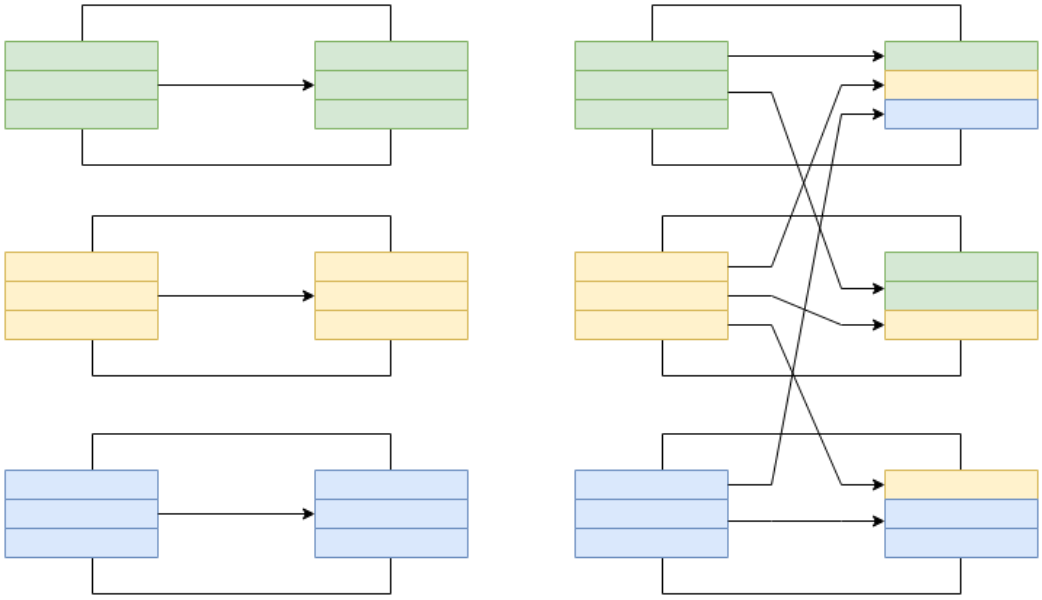
Es muy posible que los datos con los que se necesite tratar estén en diferentes objetos RDD, por lo que Spark define dos tipos de operaciones de transformación: *narrow transformation* y *wide transformation*.

- *Narrow transformation*: se utiliza cuando los datos que se necesitan tratar están en la misma partición del RDD y no es necesario realizar

una mezcla de dichos datos para obtenerlos todos. Algunos ejemplos son las funciones `filter()`, `sample()`, `map()` o `flatMap()`.

- *Wide transformation*: se utiliza cuando la lógica de la aplicación necesita datos que se encuentran en diferentes particiones de un RDD y es necesario mezclar dichas particiones para agrupar los datos necesarios en un RDD determinado. Ejemplos de wide transformation son: `groupByKey()` o `reduceByKey()`.

Una representación gráfica de ambos tipos de transformaciones es la que se puede apreciar en la figura 2.5.



(a) Narrow transformation (b) Wide transformation

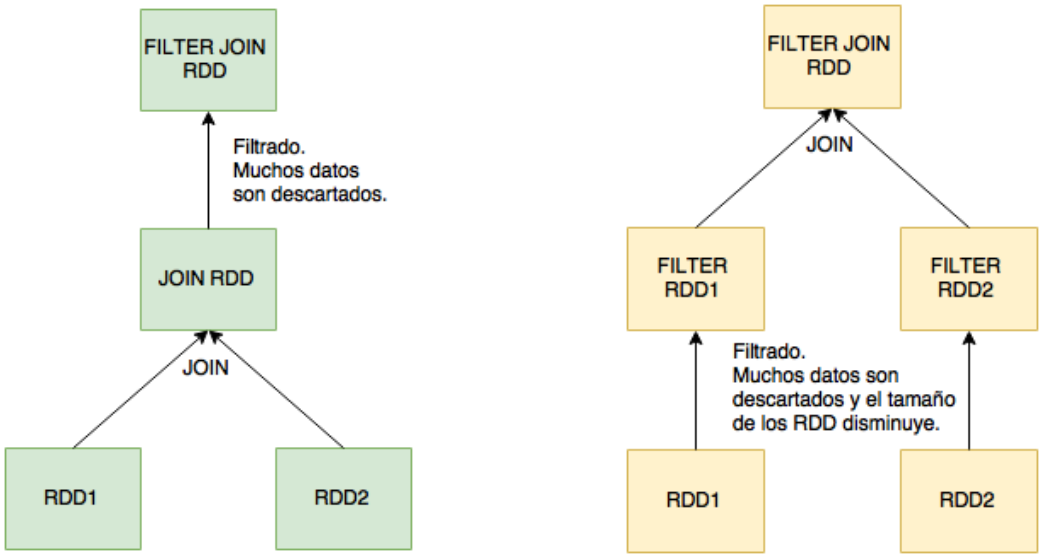
Figura 2.5: Tipos de transformaciones en objetos RDD

En algunos casos es posible realizar un reordenamiento de datos para reducir la cantidad de datos que deben ser mezclados. A continuación se muestra un ejemplo de un *JOIN* entre dos objetos RDD seguido de una operación de filtrado.

Por ejemplo, dados dos objetos RDD (RDD1 y RDD2), con variables 'a' y 'b', se va a realizar una operación de *JOIN* entre ambos conjuntos de datos para los casos en los que 'a' sea mayor que 5 y 'b' sea menor que 10:

```
SELECT a, b FROM RDD1 JOIN RDD2 WHERE a>5 AND b<10
```

Esta operación puede realizarse de dos maneras, tal y como se aprecia en la figura 2.6. La primera opción es la representada en la figura 2.6a que consiste en una implementación muy simple en la que primero se realiza el *JOIN* entre los objetos RDD y luego se filtran los datos. Sin embargo, en la segunda opción que se observa en la figura 2.6b primero se realiza el filtrado por separado en ambos RDD y luego se hace el *JOIN*.



(a) Opción 1 (b) Opción 2

Figura 2.6: Planes de ejecución para una transformación JOIN

La segunda opción es más eficiente debido a que el filtrado y posterior unión de los datos se hace por separado.

Podría decirse la mezcla o barajado de datos es la operación que más coste tiene, por lo que Apache Spark proporciona un mecanismo que genera

un plan de ejecución a partir de un DAG que minimiza la cantidad de datos que son mezclados. El plan de ejecución es el siguiente:

1. Primero se analiza el DAG para determinar el orden de las transformaciones.

2. Con el fin de minimizar el mezclado de datos, primero se realizan las transformaciones *narrow* en cada RDD.

3. Finalmente se realiza la transformación *wide* a partir de los RDD sobre los que se han realizado las transformaciones *narrow*.

2.2. Sistemas de streaming

Las tecnologías utilizadas para las funciones de envío y recepción de datos son muy importantes a la hora de analizar información en tiempo real. Por lo tanto, en esta sección se analizará el funcionamiento de los tres sistemas de streaming de datos más importantes actualmente.

2.2.1. Apache Storm

Apache Storm [15] es un sistema de computación distribuida en tiempo real y de alta disponibilidad basado en una arquitectura maestro-esclavo. Está pensado para trabajar principalmente con datos que deben ser analizados en tiempo real, como por ejemplo datos de sensores que se emiten con alta frecuencia.

En Apache Storm ninguna instancia mantiene estado y éste se delega para que sea gestionado por ZooKeeper [5]. La instancia maestro se denomina **Nimbus** y las instancias esclavo se denominan **Supervisor**.

- **Maestro**: ejecuta Nimbus y es el responsable de distribuir el código a través del cluster. Además, realiza la asignación y monitorización de las tareas en los distintos nodos que componen dicho cluster.
- **Esclavo (worker node)**: ejecuta Supervisor, que se encarga de recoger y procesar los trabajos que le son asignados en la máquina donde se ejecuta. Los nodos esclavo hacen que se distribuya el trabajo a lo largo del cluster y, en caso de que fallara uno de ellos el maestro lo sabría gracias a Zookeeper y redirigiría las tareas a otro nodo.
- **Zookeeper**: no es un componente de Apache Storm propiamente dicho, pero sí que es necesario en la arquitectura debido a que será el encargado de la coordinación entre el Nimbus y los Supervisors.

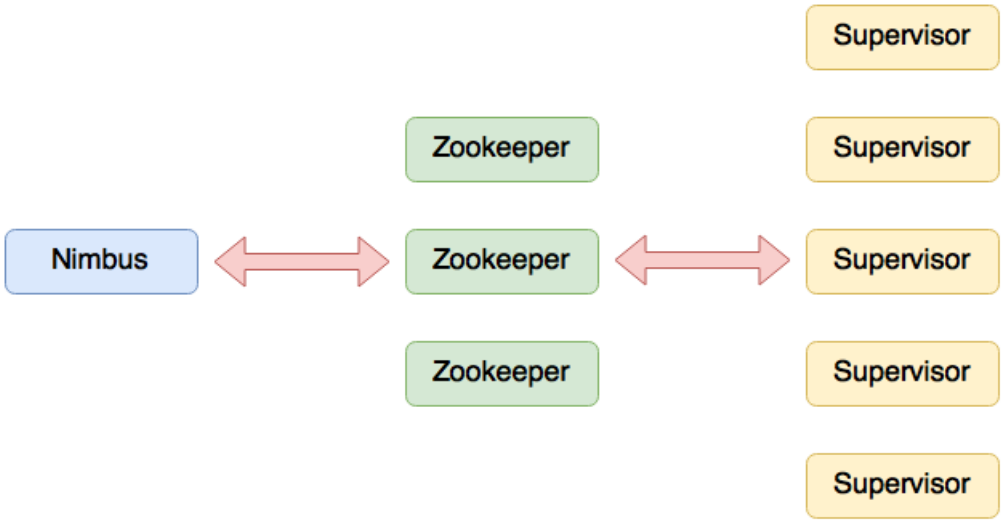


Figura 2.7: Arquitectura de Apache Storm

Apache Storm se compone de dos partes principales: Spout y Bold. La primera es la encargada de recoger el flujo de datos de entrada y la segunda procesa o transforma los datos.

Spout

El componente Spout de Apache Storm es el encargado de la ingesta de los datos en el sistema.

Bolt

El Bolt se encarga de consumir los datos que emite el Spout y los procesa en función de lo que dicte el algoritmo que esté ejecutando. Es recomendable que cada Bolt realice una única tarea y, si el programa necesita realizar varios cálculos o transformaciones sobre los datos que le llegan al Bolt, lo mejor es que se dividan en distintos Bolt para mejorar la eficiencia y la escalabilidad. Tanto el Spout como el Bolt emiten datos que serán enviados a los Spouts que estén suscritos a ese determinado stream configurado en la topología. Por ejemplo si se quiere contar las veces que aparece cada palabra en un texto, se puede hacer un Bolt que se encargue de contar las que empiezan por vocal y otro para las que empiezan por consonante. **El Spout sería el encargado de redirigir los datos a uno u otro Bolt**.

Por otra parte, **una topología es una red en la que se añaden instancias de Spouts y Bolts y que está diseñada para que escale el sistema mediante su despliegue en un cluster de Apache Storm**, que es quien se encargará de particionar los datos de entrada y redistribuirlos por los diferentes componentes. Un ejemplo es el que se puede apreciar en la figura 2.8.

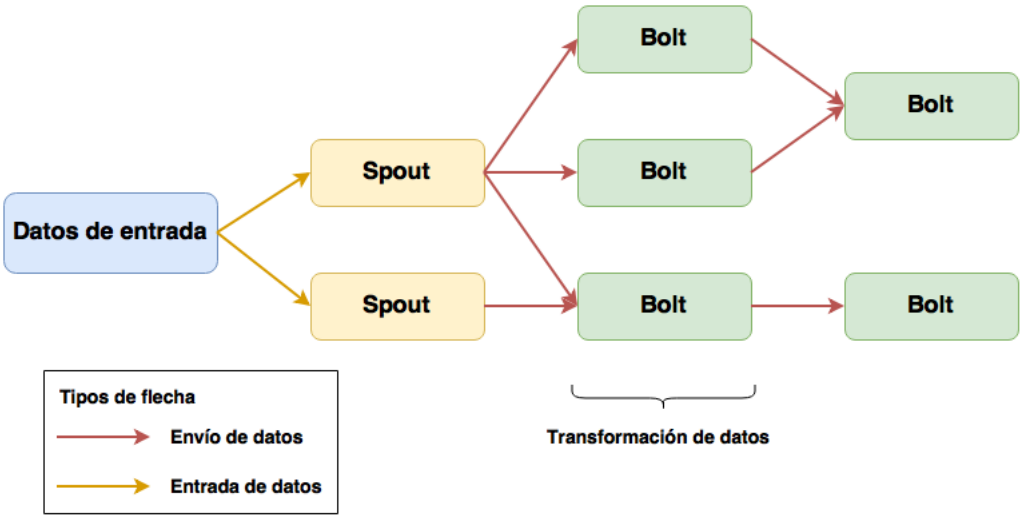


Figura 2.8: Topología con instancias de Spouts y Bolts de Apache Storm

Un cluster de Apache Storm es muy parecido a un cluster Hadoop. El equivalente al MapReduce de Hadoop sería el concepto de topología en Apache Storm. La principal diferencia es que mientras una ejecución de MapReduce termina cuando finaliza la tarea, una topología se queda esperando datos de entrada eternamente, o hasta que se mate el proceso.

2.2.2. Spark Streaming

Spark Streaming fue publicado como parte de Apache Spark 0.7 y pasó a ser *alpha* en la versión 0.9, siendo desde entonces una herramienta bastante estable que puede ser utilizada en muchos casos de uso en los que sea necesario el procesamiento de datos en tiempo real.

Los sistemas de procesamiento en *batch* como Hadoop tienen una gran latencia y no son útiles para aplicaciones que necesiten operar cerca del tiempo real. Por otra parte, Apache Storm garantiza el procesamiento de un dato de entrada **al menos una vez**, lo que puede causar inconsistencia debido a que un dato de entrada puede ser procesado dos veces.

En muchos casos se utiliza Hadoop y Apache Storm para procesamiento en batch o tiempo real, pero el uso de dos modelos de programación diferentes incrementa el tamaño del código y el número de errores, entre otras cosas. Es aquí cuando surge Spark Streaming, que ayuda a solucionar estos problemas.

Figura 2.9: Fuentes de entrada de datos en Spark Streaming

Spark Streaming puede recibir datos de diversas fuentes, como Apache Kafka, Flume, Twitter, ZeroMQ, Kinesis o sockets TCP, los cuales pueden ser procesados utilizando complejos algoritmos a los que se accede mediante funciones de alto nivel como `map`, `reduce` o `join`. Además, a los datos recibidos de Spark Streaming se le pueden aplicar algoritmos de machine learning con el propio sistema de Spark.

Figura 2.10: Spark Streaming

Internamente, Spark Streaming funciona como se muestra en la figura 2.10. Primero recibe un stream de datos y los divide en batches, los cuales son procesados por Apache Spark, que genera batches de datos ya procesados.

Por otra parte, Spark Streaming proporciona un alto nivel de abstracción

con los llamados streams discretizados o DStreams [6], que representan un stream continuo de datos. Los DStreams pueden ser creados al leer datos de las fuentes anteriormente mencionadas como Apache Kafka o Twitter, o también aplicando operaciones a otros DStreams. Internamente un DStream se representa como una secuencia de RDDs.

Spark Streaming procesa los datos en tiempos de batch, también llamados ventanas. Suponiendo cuatro intervalos de tiempo desde el cero hasta el cuatro, tal y como se ve en la figura 2.11, habrá cuatro RDDs, cada uno con su contenido.

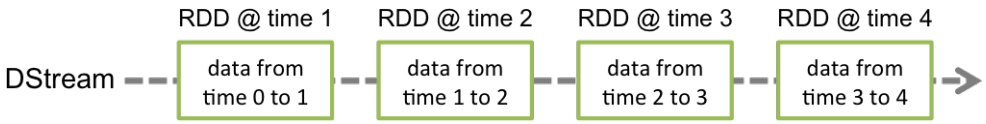


Figura 2.11: Contenido de un DStream en 4 intervalos temporales

Cualquier operación aplicada a un DStream se transforma en una operación aplicada a los RDDs que contiene internamente. Por ejemplo, si cada RDD tiene diferentes líneas de un texto que deben ser contadas, se aplicarán las funciones de recuento de palabras a cada RDD del DStream, como se ve en la figura 2.12.

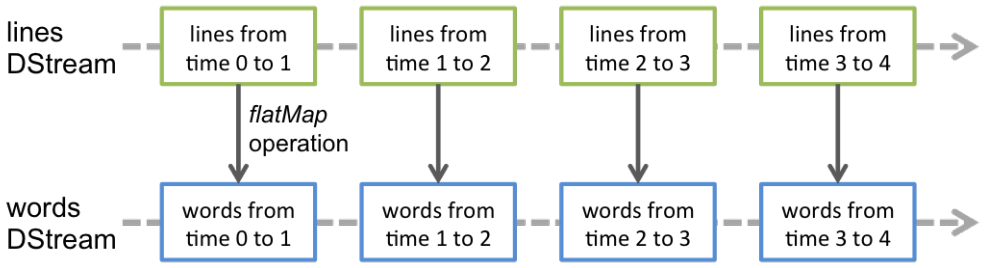


Figura 2.12: Contar palabras contenidas en líneas de un DStream

Las transformaciones que se realizan en un RDD se aplican internamente por Apache Spark. Las operaciones ejecutadas en un DStream son transparentes al programador gracias a la API de alto nivel con la que se desarrolla el código.

El soporte para Python de Spark Streaming se introdujo en la versión 1.2, pero a día de hoy no tiene las funcionalidades completas, al contrario que Scala y Java.

2.2.3. Apache Kafka

Apache Kafka [2] es un sistema de mensajería distribuido basado en publicación-suscripción. Fue desarrollado originalmente por LinkedIn y más tarde pasó a formar parte de la Apache Software Foundation. Se diferencia de los sistemas de mensajería tradicionales en varios puntos: está diseñado para ser escalable por naturaleza, ha sido desarrollado en Scala, ofrece un alto rendimiento tanto en publicación como suscripción y soporta más de una suscripción y balanceo automático en caso de fallo, además de persistencia de mensajes en disco.

Arquitectura

Para conocer los conceptos básicos de Apache Kafka conviene realizar una explicación de su arquitectura, que contiene los siguientes componentes:

- Un streaming de mensajes de un tipo determinado es conocido como topic. Un mensaje se define como una carga de bytes y un topic es una categoría en la que los mensajes son publicados.
- Un productor puede ser cualquiera con la posibilidad de publicar mensajes en un topic.
- Los mensajes publicados son almacenados son almacenados en un conjunto de servidores llamados brokers o clusters Kafka.
- Un consumidor puede suscribirse a uno o más topics y leer los mensajes publicados desde los brokers.

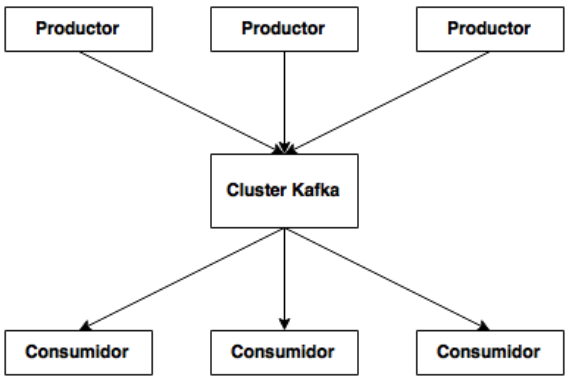


Figura 2.13: Funcionamiento de Apache Kafka.

El productor tiene la posibilidad de escoger el método de serialización más conveniente para enviar el contenido del mensaje. A continuación se verán algunos ejemplos de snippets de código en Java para ver cómo enviar y

recibir mensajes.

Clase básica para crear un productor:

```
class Producer():
    def run(self):
        client = KafkaClient("KAFKA_IP_ADDRESS:PORT")
        producer = SimpleProducer(client)
        producer.send_messages("my-topic", "message")
```

En este código se ha definido la clase **Producer**, utilizada para crear un productor. Contiene el método **run** para ejecutar el código que crea el productor y manda un mensaje 'message' a un topic 'my-topic'.

Clase básica para crear un consumidor:

```
class Consumer():
    def run(self):
        client = KafkaClient("KAFKA_IP_ADDRESS:PORT")
        consumer = SimpleConsumer(client, "my-topic")
        for message in consumer:
            print(message)
            # Do Something with the message
```

Este código define la clase **Consumer** para crear un consumidor y contiene, igual que el anterior, un método **run** para leer los mensajes del topic 'my-topic'. Tras leer los mensajes, puede hacer cualquier operación.

En la figura 2.14 se aprecia una visión general de la arquitectura de Apache Kafka. Dado que este sistema de mensajería es distribuido por naturaleza, un cluster Apache Kafka suele estar formado por múltiples *brokers*. El balanceo de carga es de gran importancia en Apache Kafka, por lo que escribe en un *topic* dividido en diferentes particiones que permiten paralelizar la escritura, logrando así una mejor eficiencia.

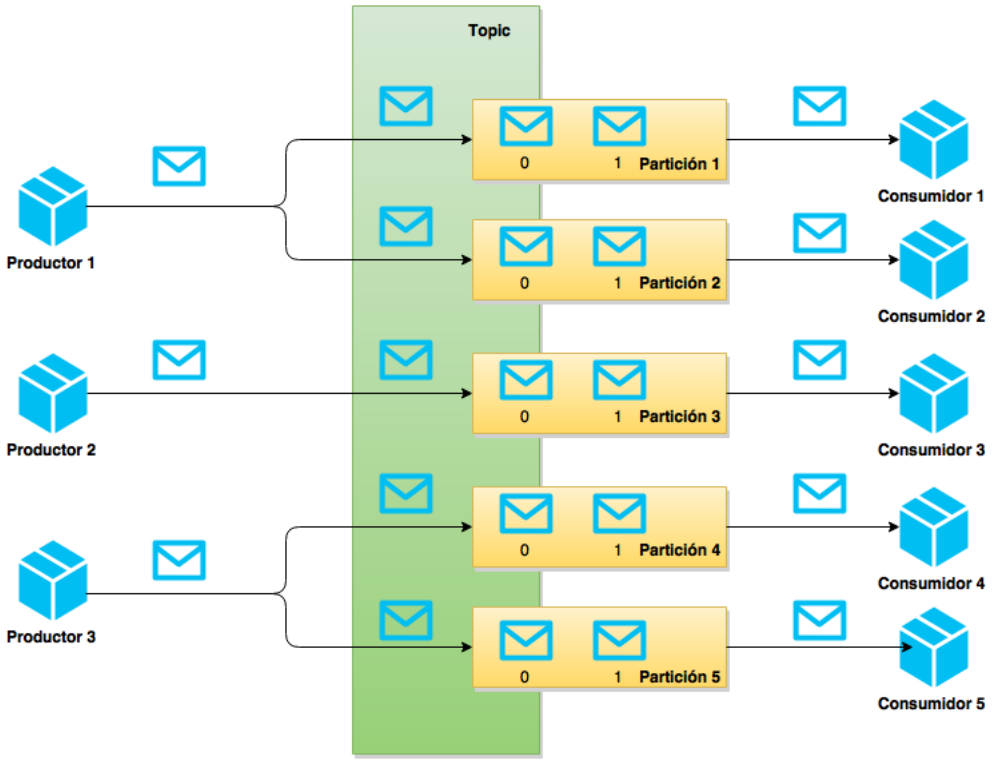


Figura 2.14: Arquitectura de Apache Kafka.

Almacenamiento

La lógica de almacenamiento en Apache Kafka es muy simple. Cada partición de un topic se corresponde con un log. Físicamente, un log se implementa como un conjunto de archivos del mismo tamaño. Cada vez que un productor publica un mensaje en una partición, el broker simplemente añade dicho mensaje al final del conjunto de mensajes del topic correspondiente. Una vez se ha llegado a un determinado y configurable número de mensajes, se envían al consumidor.

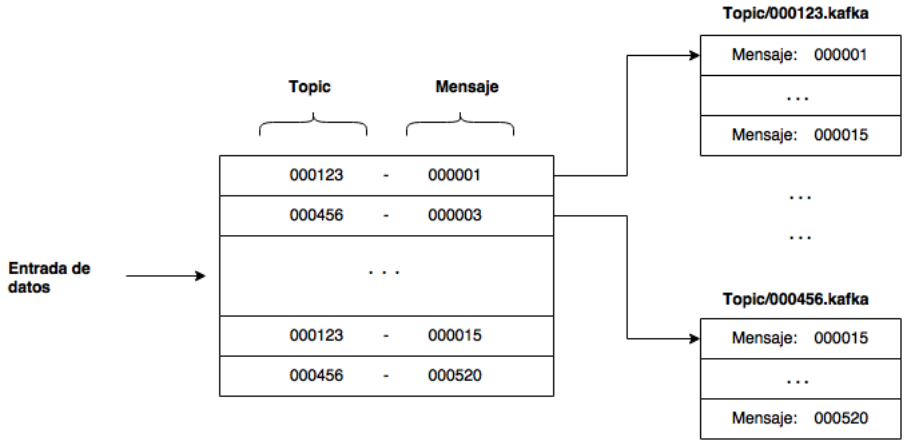


Figura 2.15: Almacenamiento en Apache Kafka.

Broker

Al contrario que con otros sistemas de mensajería, los brokers de Apache Kafka no tienen estados, es decir, no saben si un mensaje ha sido leído o no. Toda la lógica y gestión de los mensajes la lleva el consumidor.

Es complicado saber cuándo eliminar un mensaje en el broker, por el hecho de que éste no tiene conocimiento del estado del consumidor. Debido a esto y simplificando las cosas, Apache Kafka resuelve este problema eliminando los mensajes que hayan estado en el broker una determinada cantidad de tiempo. Gracias a esto, un consumidor puede volver atrás en el tiempo y consumir de nuevo datos que ya había leído.

2.3. Lenguajes de programación

El preprocesamiento de los datos es el primer paso a realizar en cualquier proyecto de ciencia de datos y normalmente el más importante. Los científicos de datos suelen preferir programar con herramientas como las que proporcionan R o Python. Sin embargo, cuando se necesitan utilizar herramientas

Big Data como Apache Spark, puede ser necesario cambiar de lenguaje de programación.

2.3.1. Scala y Java

La mayoría de las aplicaciones son programadas en el lenguaje Scala [11] o Java [12] porque implica usar el lenguaje de programación nativo de Spark. Programar en Scala o Java tiene varias ventajas al hacer ciencia de datos:

- **No hay que convertir código.** Si se ejecuta código en R o Python sobre la máquina virtual Java (JVM) en la que se basa Scala hay que realizar un trabajo extra para convertir tanto el código como los datos a otro entorno. Al llevar a cabo este proceso de conversión, puede que haya cosas que se queden en el camino o que no funcionen como era de esperar. Si, por lo contrario, se desarrollan los algoritmos con la API de Scala o Java, el programador puede estar seguro de que el código va a funcionar tal y como se espera.

- **Tener acceso a las últimas funcionalidades.** Todas las librerías de Spark para machine learning, procesado de streams de datos o visualización están escritas en Scala. Por lo tanto, si se desarrolla en Scala, tendremos soporte siempre para las últimas funcionalidades. Al programar en R o Python se tardará en tener soporte, ya que hay que hacer doble trabajo implementando las librerías correspondientes.

2.3.2. Python

Python es un lenguaje de programación creado por Guido van Rossum. Es un lenguaje multiparadigma, ya que permite varios estilos de programación:

orientada a objetos, imperativa y funcional. La ventaja de desarrollar aplicaciones de Apache Spark en este lenguaje es que es más sencillo de utilizar que Scala, pero su principal punto débil es que va un paso atrás en cuanto a frecuencia de actualizaciones de sus librerías y funcionalidades en comparación con Scala y Java, lo que hace que no siempre sea la opción adecuada.

La versión de Apache Spark que se programa con la API de Python se denomina PySpark [10].

2.4. Resumen

En este capítulo se han analizado las tecnologías habilitantes que forman parte del ecosistema Big Data. Los sistemas más importantes son Apache Hadoop y Apache Spark y se ha explicado su funcionamiento a nivel interno. Por otra parte, se han mostrado las herramientas de streaming de datos que conectan las fuentes de datos externas con Hadoop o Spark. Para finalizar, se han expuesto los diferentes lenguajes de programación con los que se desarrollan aplicaciones Big Data.

Capítulo 3

Arquitectura de la plataforma

3.1. Requisitos

A continuación se realizará un listado de los requisitos del proyecto para poder llevar a cabo su desarrollo:

- **Datos:** los datos son la parte más importante a la hora de realizar tareas de machine learning y predicción. Las fuentes de datos son las encargadas de enviar la información al sistema de tratamiento masivo de datos que se propone en este capítulo. La estructura y formato de los datos no tiene que ser siempre la misma, ya que dependerá del caso de estudio.

- **Streaming:** debido a que no siempre se va a tener un fichero con los datos guardados en el sistema es necesario habilitar una entrada de datos por streaming cuyo origen será una fuente externa, la cual enviará los datos mediante alguna de las herramientas de streaming analizadas en el capítulo anterior.

- **Análisis de los datos:** los datos recibidos deben ser analizados y

preprocesados antes de aplicar técnicas de machine learning sobre ellos. Un mal formato de los datos puede causar predicciones erróneas en la solución del problema.

- **Entrenamiento y creación de modelos:** es necesario que se puedan crear modelos para realizar las predicciones. El algoritmo utilizado dependerá del caso de estudio.
- **Prueba de modelos:** los modelos deben ser puestos a prueba para ver si cumplen con los objetivos planteados. Existen métricas que indican lo efectivo que es un modelo, las cuales serán utilizadas por el científico de datos en función del caso de estudio.
- **Almacenamiento de modelos:** con el fin de no estar siempre entrenando el modelo cada vez que llega un dato, dicho modelo se debe serializar para más tarde poder importarlo en cualquier aplicación.

3.2. Casos de uso

En esta sección se mostrarán dos casos de uso diferentes que se aplicarán a cada caso de estudio. El primero es el entrenamiento de un modelo y el segundo es el de realización de predicciones en base al modelo previamente entrenado.

3.2.1. Entrenamiento de un modelo

Descripción: el sistema deberá comportarse como se describe en el siguiente caso de uso cuando reciba datos de entrenamiento.

Actor: científico de datos.

Actividad: el científico de datos introduce datos de entrenamiento de un problema determinado dentro del sistema de predicción. El sistema los recibe y crea un modelo de un determinado algoritmo especificado por el científico de datos. El modelo se serializa para poder realizar predicciones sin necesidad de ser entrenado cada vez que se recibe un dato.

Flujo de eventos:

1. **Científico de datos:** recoge datos y crea un conjunto de datos.
2. **Científico de datos:** introduce el conjunto de datos en el sistema.
3. **Sistema de predicción:** recibe los datos y los formatea para hacerlos compatibles con el predictor.
4. **Científico de datos:** divide los datos en entrenamiento y test.
5. **Sistema de predicción:** entrena los datos y crea un modelo.
6. **Sistema de predicción:** prueba el modelo.
7. **Sistema de predicción:** serializa el modelo.

Excepciones: si el científico de datos determina que el modelo generado por el sistema no es lo suficientemente bueno, se vuelve al paso 5 utilizando otros parámetros de configuración y se repite el proceso desde ese punto.

3.2.2. Realización de una predicción

Descripción: el sistema deberá comportarse como se describe en el siguiente caso de uso cuando reciba datos de una fuente externa.

Actor: fuente externa de datos.

Actividad: una fuente externa envía información sobre su estado al predictor. Al completar el envío de datos, se realiza una predicción. Por medio de una interfaz gráfica o línea de comandos el predictor muestra el resultado obtenido.

Precondición: el modelo utilizado por el predictor debe haber sido entrenado y serializado.

Flujo de eventos:

1. **Sistema de predicción:** abre una conexión para recibir datos.
2. **Fuente externa de datos:** envía datos.
3. **Sistema de predicción:** formatea los datos para hacerlos compatibles con el predictor.
4. **Sistema de predicción:** analiza los datos y genera una predicción.
5. **Sistema de predicción:** muestra la predicción por pantalla.

3.3. Criterios de selección

En esta sección se expondrán los criterios de selección que se han tenido en cuenta para elegir la tecnología más conveniente en cada módulo de la arquitectura propuesta.

3.3.1. Apache Spark vs. Apache Hadoop MapReduce

Conviene dejar claro que Apache Spark no es una alternativa a Apache Hadoop, sino a Apache Hadoop MapReduce. Apache Spark no solo puede utilizar HDFS como fuente y sumidero de datos, sino que puede ejecutarse en Apache Hadoop YARN. Por lo tanto, Apache Spark es una alternativa a MapReduce dentro del ecosistema Hadoop.

A continuación, se comentarán varios puntos que se consideran de gran importancia a la hora de elegir un sistema u otro.

Velocidad de proceso

Apache Spark realiza operaciones entre 10 y 100 veces más rápido que el framework Hadoop MapReduce simplemente reduciendo el número de lecturas y escrituras que se realizan en disco.

En Spark, el concepto de los RDD (*Resilient Distributed Datasets*) permite almacenar datos en memoria y reservar el disco solo para cuando sea necesario. Por lo tanto, la ejecución de una aplicación en Spark es mucho más rápida que en Hadoop MapReduce por el eficiente uso de la memoria que implementa.

A finales de noviembre de 2014 Apache Spark participó en la competición Sort Benchmark [13] en la categoría Daytona Gray, que consiste en ver cómo de rápido se pueden procesar 1 trillón de registros, sumando un total de 100 TB de datos. El récord lo estableció Yahoo en 72 minutos utilizando un cluster de Hadoop MapReduce de 2100 nodos. Sin embargo, Spark ha batido dicho récord con 23 minutos utilizando un cluster formado por 206 nodos, es decir, Spark ha sido 3 veces más rápido utilizando una décima parte de máquinas.

Además, ya que no existe una categoría con 1 PB de datos, se realizaron pruebas externas a la competición con 10 trillones de registros y con 190 nodos se tardó 4 horas en Spark, mientras que con Hadoop MapReduce se tardó 16 horas haciendo uso de 3800 máquinas.

Fácil gestión

Con Apache Spark es posible realizar tareas de *streaming*, *batch processing* y *machine learning* en el mismo cluster. Sin embargo, con Hadoop hay que utilizar diferentes tecnologías y unificarlas en un mismo proyecto, ya que no están todas integradas en la plataforma, al contrario que en Apache Spark.

Procesamiento de streams en tiempo real con Spark Streaming

Con Apache Hadoop MapReduce es posible procesar grupos de datos almacenados, pero Spark va más allá e incluso permite modificar y procesar datos en tiempo real. Spark Streaming se basa en *streams* discretizados, que propone un nuevo modelo para realizar computación de ventana utilizando *micro batches*. Hadoop no soporta esta funcionalidad.

Latencia

Una de las ventajas de Spark es su baja latencia computacional, la cual consigue gracias a que guarda en memoria resultados parciales de operaciones a través de sus nodos. Por otro lado, Hadoop MapReduce está completamente orientado a disco.

Recuperación

El RDD es la principal abstracción de Apache Spark. Permite la recuperación ante fallos en nodos gracias a que recalcula el DAG, además de soportar

un método de recuperación similar al de Hadoop que se basa en el uso de *checkpoints*.

Gestionar la ejecución de aplicaciones con un DAG permite a Spark optimizar dicha ejecución, tanto en rapidez como en tolerancia a fallos y recuperación ante los mismos.

Planificación

Apache Spark funciona como su propio planificador del flujo de ejecución. Sin embargo, Hadoop MapReduce necesita un planificador externo como Oozie [8] o Azkaban [9] para flujos complejos.

Coste

La memoria de un cluster Apache Spark debe ser al menos tan grande como la cantidad de datos que se nececesita procesar, ya que como se ha comentado anteriormente, no se almacenará en disco sino en dicha memoria para un rendimiento óptimo. Sin embargo, Spark necesita menos máquinas para realizar las mismas operaciones que Hadoop MapReduce.

	Hadoop MapReduce	**Apache Spark**
Rapidez	No	Sí
Fácil gestión	No	Sí
Streaming	No	Sí
Caché	No	Sí
Planificador	Externo	Interno
Coste	Alto (más máquinas)	Bajo (menos máquinas)

Cuadro 3.1: Comparativa Apache Hadoop MapReduce y Apache Spark

3.3.2. Spark Streaming vs. Apache Storm

Modo de procesamiento y latencia

Apache Storm es un motor de streaming de datos en verdadero tiempo real y que puede realizar *micro-batching*, mientras que el Spark Streaming es un motor de procesamiento de streams en *batch* con *micro-batches*, pero no realiza streaming en tiempo real en el más estricto sentido de la palabra.

Procesamiento en *batch*

El procesamiento en *batch* o por lotes es un concepto familiar en el procesado masivo de streaming de datos. El tamaño del *batch* puede ser grande o pequeño y consiste en ejecutar tareas en grupo una vez se ha llegado al número máximo establecido. Esto difiere del procesamiento en tiempo real en que los datos se almacenan, no se procesan según llegan, por lo que puede haber un pequeño retardo.

3.3.3. Lenguaje de programación

Para elegir un lenguaje de programación se va a utilizar la tabla 3.2, en la que se realiza una comparativa de las principales características.

	Scala	**Java**	**Python**
Nativo	Sí	Sí	No
Rapidez	Alta	Alta	Normal
Actualización de bibliotecas	Frecuente	Frecuente	Poco frecuente
Curva de aprendizaje	Media	Media	Muy rápida

Cuadro 3.2: Comparativa de los lenguajes de programación

Apache Spark está desarrollado en Scala y proporciona APIs para los tres lenguajes. Si se necesita entender cómo funciona o modificar un fragmento de

código de Spark, Scala es la opción correcta para programar aplicaciones. Por otra parte, hay que tener en cuenta la curva de aprendizaje de cada lenguaje. La de Python es bastante rápida en comparación con los otros dos debido a su sencillez. Sin embargo, esto es totalmente secundario cuando se trata de desarrollar un programa con las últimas funcionalidades. Python siempre va a ser el último que va a recibir soporte, mientras que Scala y Java van a la par en cuanto a actualizaciones y desarrollo.

Para evaluar de forma prácticas sus capacidades, los dos primeros casos de estudio han sido desarrollados en Python por su simplicidad. Sin embargo, en el tercero se ha utilizado Scala debido a que la librería que necesita el código para poder ejecutarse no está implementada en la última versión de Apache Spark, que a día de hoy es la 1.3.

3.4. Arquitectura

La arquitectura propuesta está formada por diversos agentes:

- **Monitorización:** de la parte de monitorización se encargan servidores externos a nuestro sistema. Estos servidores son los encargados de enviar los datos por streaming para poder leerlos y tratarlos en pasos siguientes. Los datos enviados y sus formatos dependerán del problema que se esté solucionando. Esta parte es la que hace que se cumpla el requisito de datos.

- **Entrada de datos con Apache Kafka:** Apache Kafka ha sido el elegido como vía de entrada de datos en streaming debido a su diseño basado en publicación-suscripción. Facilita la conexión entre Apache Spark y el bloque de monitorización, permitiendo la recepción de datos

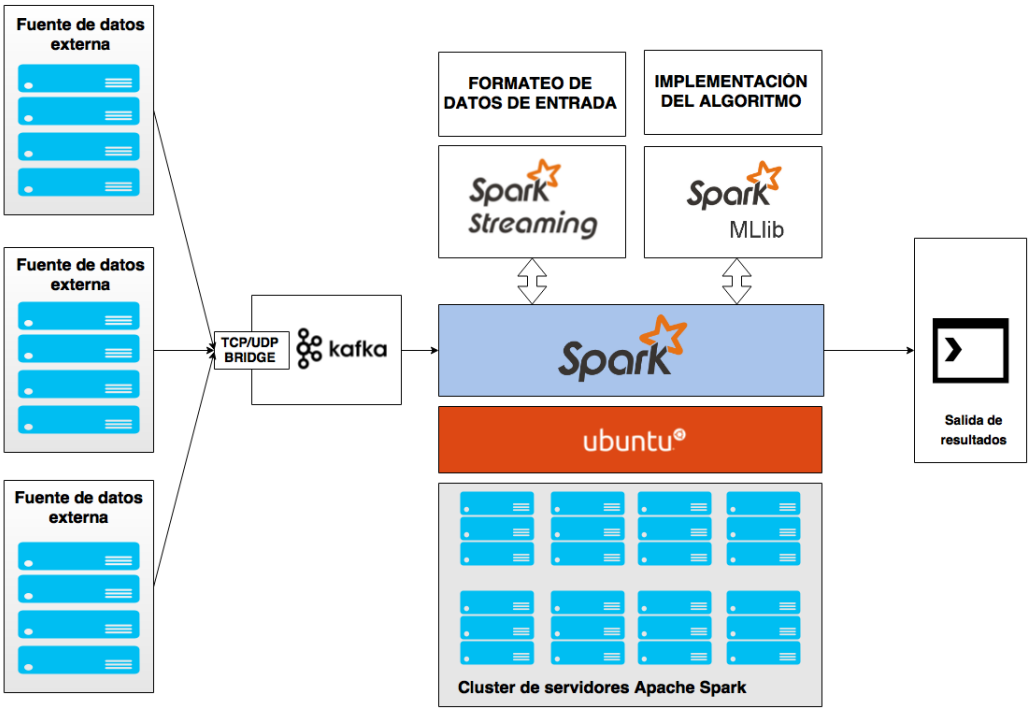


Figura 3.1: Arquitectura de la solución.

tanto por TCP como por UDP. Gracias a esto, se cumple el requisito de streaming en la arquitectura.

- **Spark Streaming (opcional):** Spark Streaming es opcional porque dependerá del caso de estudio. También es posible utilizar Apache Storm, pero su integración con Spark no es directa, lo que sí ocurre con Spark Streaming.

- **Procesado de datos:** este bloque se divide en dos.

  - **Parseado de datos:** los datos llegarán con un formato que puede ser distinto del que se necesita utilizar para un determinado algoritmo, por lo que deben ser parseados al formato correcto.

  - **Análisis de datos:** haciendo uso de las librerías de machine learning que proporciona Apache Spark será posible crear, entrenar y

probar modelos con los datos recibidos.

- **Salida de datos:** en este caso la salida del resultado de aplicar un determinado algoritmo a un conjunto de datos será por consola, pero también es posible escribir en fichero o crear una aplicación web en la que mostrar los resultados de forma visual.

La arquitectura puede parecer sencilla a simple vista, pero en realidad no lo es tanto. El bloque que contiene las funciones de parseado y análisis de datos estará montado en un servidor que esté ejecutando Apache Spark. Sin embargo, para sacar el máximo provecho de esta plataforma conviene tener muchos servidores, lo cual se conseguirá mediante un cluster de servidores de Spark.

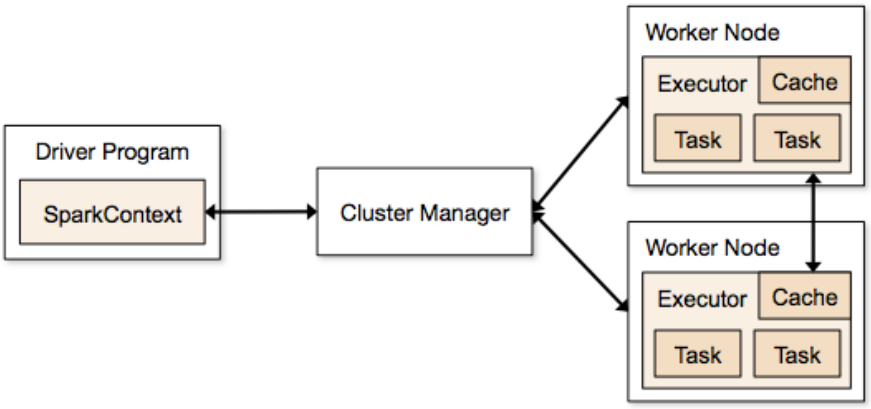


Figura 3.2: Distribución de un cluster.

En la figura 3.2 se muestra la arquitectura de un cluster montado con Apache Spark y se puede observar que existen distintos agentes. El primero de ellos es el controlador. Tras esto, se encuentra el cluster manager y finalmente los workers. De una manera muy resumida, el manager hace las veces de maestro y el worker de esclavo. Conviene conocer algunos términos:

- **Aplicación**: programa desarrollado sobre Spark por un programador.

- **Controlador del programa (driver program)**: es el proceso que ejecuta la función de inicio (`main`) de la aplicación.
- **Cluster manager**: servicio externo utilizado para gestionar los recursos de los workers.
- **Worker**: nodo (servidor) que puede ejecutar código Spark.
- **Task**: operaciones (tareas) a realizar por un worker.

Para montar un cluster de servidores que ejecuten Spark es necesario instalar Spark en cada uno de ellos. Además, existen varios archivos de configuración para determinar qué nodo es el maestro y qué nodos son los esclavos. Además, es posible configurar los recursos hardware de los que puede disponer el maestro en cada nodo esclavo.

En el apéndice 1 está disponible una guía para montar un cluster de varias máquinas.

3.5. Resumen

En este capítulo se han establedido los requisitos que debe cumplir un ecosistema big data para análisis masivo de datos con machine learning. Además, se plantean dos casos de uso aplicados a la arquitectura propuesta en la sección 3.4. A continuación, se han realizado comparaciones entre diferentes tecnologías y sus alternativas y finalmente, se ha explicado cómo es el diseño de la arquitectura planteada para los casos de estudio del capítulo siguiente, la cual hace uso de las tecnologías analizadas a lo largo del trabajo.

Capítulo 4

Casos de estudio

En este capítulo se presentan tres casos de estudio diferentes. Se ha optado por implementar varios ejemplos con el objetivo de mostrar que la arquitectura propuesta en la sección 3.4 es válida para cualquier caso en el que se tengan que realizar tareas de machine learning sobre datos recibidos de una fuente externa, ya sea una red social como Twitter o información de estado de servidores.

4.1. Detección de anomalías en redes

El objetivo de la detección de anomalías en redes es, tal y como su nombre indica, encontrar comportamientos inusuales en una red. En este caso, se va a realizar un sistema de detección de intrusos. Mediante el análisis del tráfico de una red en tiempo real, un NIDS (Network Intrusion Detection System) busca anomalías como ataques DoS (Denial of Service), escaneadores de puertos o intentos de accesso a un ordenador. Inspeccionando los paquetes es posible detectar patrones sospechosos, pero no únicamente los paquetes entrantes, sino también los salientes o el tráfico local. Además, no solo se

pueden detectar anomalías en una red, sino también en ataques basados en Web [16].

Es complicado realizar una configuración óptima de un NIDS, ya que si no se hace correctamente, es posible obtener muchos falsos positivos y muchos ataques podrían pasar inadvertidos.

4.1.1. Conjunto de datos

El conjunto de datos que se ha utilizado para el experimento contiene 4.9 millones de registros de conexiones, sumando un total de 748 MB. Es un archivo grande, pero no demasiado. Sin embargo, tiene los registros suficientes para realizar el análisis.

Por cada conexión, el conjunto de datos [17] contiene información como el número enviado de bytes, los intentos de login o errores TCP, entre otras cosas, sumando un total de 38 parámetros.

En este caso, los ataques se dividen en cuatro categorías:

- **DOS (denial-of-service)**: ataques de denegación de servicio.
- **R2L (remote-to-local)**: acceso remoto no autorizado.
- **U2R (user-to-root)**: intento de acceso como superusuario.
- **Probing**: escaneo de puertos.

A continuación se pueden observar los dos primeros registros del conjunto de datos:

- `0,tcp,http,SF,215,45076,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,`
  `0.00,0.00,0.00,0.00,1.00,0.00,0.00,0,0,0.00,0.00,0.00,0.00,`
  `0.00,0.00,0.00,0.00,normal.`

- `0,tcp,http,SF,162,4528,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,2,2,0.00,0.00,0.00,0.00,1.00,0.00,0.00,1,1,1.00,0.00,1.00,0.00,0.00,0.00,0.00,0.00,normal.`

El primero de los registros anteriores representa una conexión TCP a un servicio HTTP, en la que se han enviado 215 bytes y se han recibido 45706 bytes. En el segundo se observa que el tipo de conexión es la misma, pero que los bytes enviados han sido 162 y los recibidos 4528.

Muchas de las columnas de estos datos toman el valor de 0 ó 1, indicando la presencia o ausencia de un parámetro. En la última columna, encontraremos el tipo de conexión, en el registro anterior se trata de una conexión normal, igual que la mayoría de los registros existentes. También se han identificado algunos casos de ataques. Esto es útil para aprender si una conexión es buena o mala, normal o ataque. Sin embargo, lo que se necesita conocer es la detección de anomalías y encontrar ataques que no sean conocidos previamente.

4.1.2. Algoritmo utilizado

Los algoritmos de machine learning [18] de clasificación y regresión son muy útiles cuando los científicos de datos saben lo que están buscando y tienen un conjunto de datos con información sobre ello, es decir, cuando se utiliza aprendizaje supervisado. En el proceso de aprendizaje supervisado se tiene un valor de salida por cada entrada. Sin embargo, hay problemas en los que el resultado es desconocido para cualquier valor de entrada. Es en estos casos en los que se utilizan técnicas de aprendizaje no supervisado, las cuales ayudan a aprender e inspeccionar automáticamente la estructura de los datos, así como a agruparlos.

- **KMeans clustering**: K-Means clustering [20] es uno de los algoritmos de clusterización o agrupamiento más utilizados. Su objetivo es detectar k clusters en n grupos de datos, donde k es un valor dado por el científico de datos que dependerá del conjunto de datos disponible. Cada observación pertenece al grupo más cercano a la media.

4.1.3. Implementación

Al realizar un proyecto de machine learning es extremadamente importante conocer los datos de los que disponemos. No solo su formato, sino su contenido. Debido a esto, el primer paso realizado ha sido inspeccionar los tipos de ataques que contiene el conjunto de datos y contar el número de veces que aparece cada uno de ellos.

```
1  print "Counting all different labels"
2  labels = raw_data.map(lambda line: line
3                                       .strip()
4                                       .split(",")[-1])
5  label_counts = labels.countByValue()
6  lblsorted = OrderedDict(sorted(label_counts.items(),
7                                 key=lambda t: t[1],
8                                 reverse=True))
9  for label, count in lblsorted.items():
10     print label, count
```

Fragmento de código 4.1: Obtener los tipos de ataques

Tras obtener los tipos de ataques y las veces que aparecen, se pueden graficar, obteniendo como resultado la figura 4.1, donde se puede comprobar que hay tres tipos de conexiones principales:

- **Ataques *smurf***: 2.807.886
- **Ataques *neptune***: 1.072.017
- **Conexiones normales**: 972.781

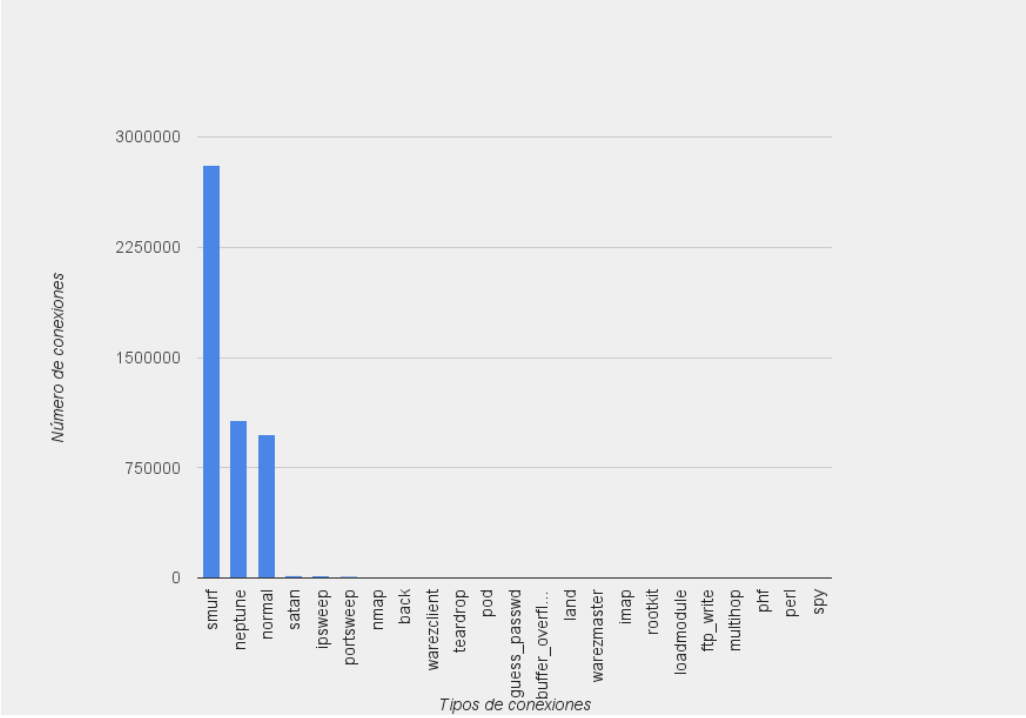


Figura 4.1: Tipos de conexiones y número de apariciones.

Esto quiere decir que hay varios tipos de ataques y que en total suman más las conexiones malas que normales.

Entrenamiento del modelo

Es en esta parte del problema en la que se pone en práctica el caso de uso de entrenamiento de un modelo propuesto en la sección 3.2.1. En este caso de estudio, el lenguaje de programación escogido ha sido Python. Además, se ha hecho uso de la librería MLLib de Apache Spark, que contiene un método para aplicar el algoritmo KMeans a un conjunto de datos.

Los parámetros de entrada de la función para entrenar el modelo son varios:

- Datos: el conjunto de datos introducidos debe ser de tipo vector.

- Número de iteraciones sobre los datos.
- Número de veces que se ejecuta.
- Tipo de inicialización.

Con esto, es posible crear un modelo del algoritmo KMeans:

```
from pyspark.mllib.clustering import KMeans
model = KMeans.train(data,
                     k,
                     maxIterations=10,
                     runs=5,
                     initializationMode="random")
```

Fragmento de código 4.2: Creación de un modelo KMeans

El fragmento de código anterior muestra cómo tras importar la librería KMeans del módulo de *clustering* de MLLib es posible crear un modelo simplemente ejecutando la función `train()`.

El problema siguiente es determinar el valor de K para una configuración eficiente del modelo. Al probar con un valor de K igual a 2 se obtienen unos resultados muy malos. Esto es debido a que el conjunto de datos es muy grande y no es un problema en el que únicamente haya dos tipos de conexiones, sino que hay muchas más. En este caso y con este valor de K, todos los puntos excepto uno quedan en el cluster número 0, por lo que no se resuelve el problema.

Como solución a esto, es posible ejecutar la simulación para distintos valores de K en un bucle. Además, para ver cuál es el mejor valor de K, se calculará la distancia media de los puntos a sus centroides correspondienes.

```
def distCentroid(datum, clusters):
    cluster = clusters.predict(datum)
    centroid = clusters.centers[cluster]
    return sqrt(sum([x**2 for x in (centroid-datum)]))
```

Fragmento de código 4.3: Función de cálculo de distancia a centroides

La función `clusteringScore()` calcula la distancia media por cada valor de K y devuelve el resultado:

```
def clusteringScore(data, k):
    clusters = KMeans.train(
                          data,
                          k,
                          maxIterations=10,
                          runs=5,
                          initializationMode="random")
    result = (k,
              data.map(lambda datum: distCentroid(
                                                  datum,
                                                  clusters))
              .mean())
    return result
```

Fragmento de código 4.4: Calcular la distancia media a los centroides en función de K

Para ejecutarla con un rango de valores de K desde 5 hasta 115 con un intervalo de 10, se ejecuta la siguiente función:

```
scores = map(lambda k: clusteringScore(values, k),
                         range(5,115,10))
```

Fragmento de código 4.5: Cálculo del valor óptimo de K

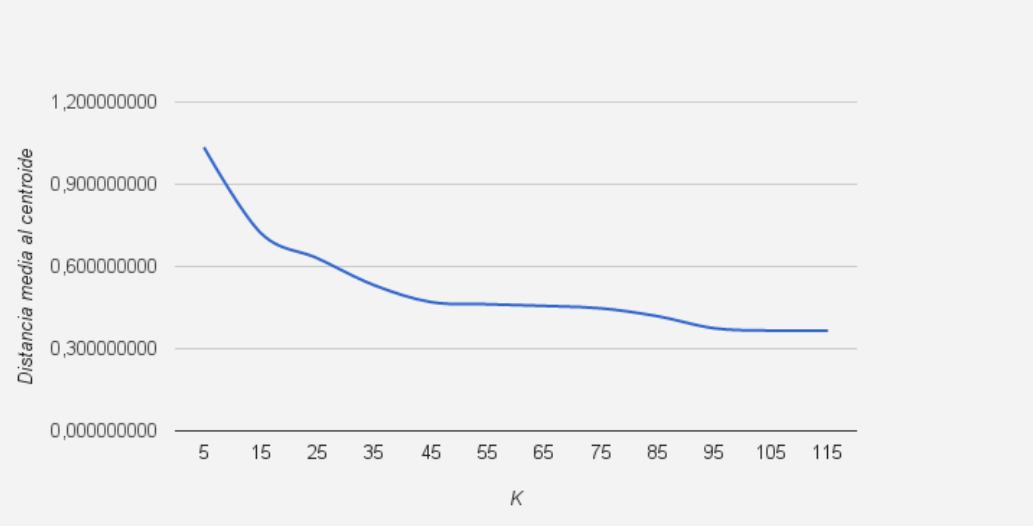


Figura 4.2: Distancia media a los centroides en función de K.

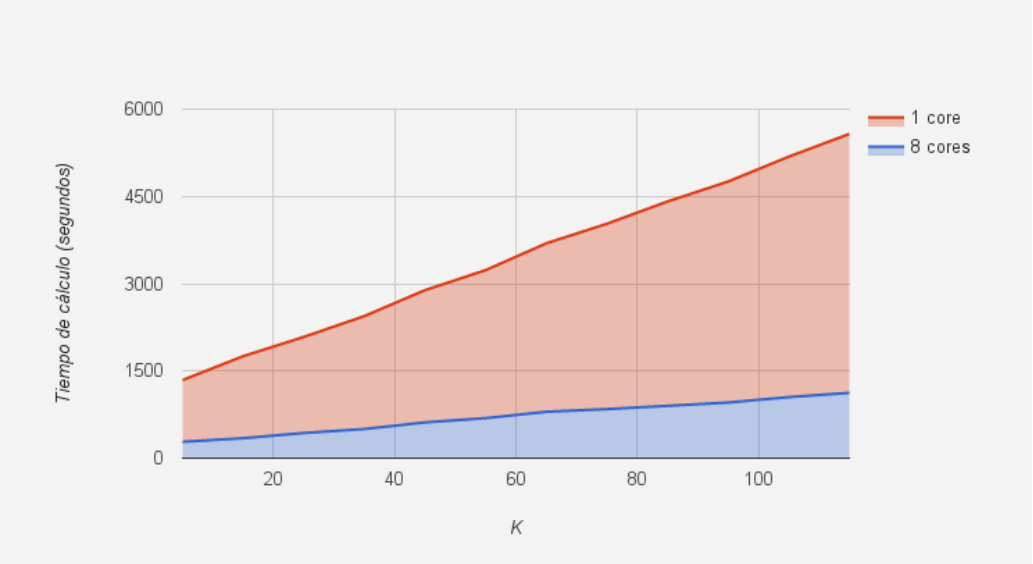


Figura 4.3: Tiempo de cálculo de los modelos en función de K y los cores utilizados.

Tal y como se puede apreciar en la figura 4.2, el valor de K óptimo es de 115, ya que es el que ofrece una menor distancia media a los centroides. A partir de K = 45 practicamente se estabiliza dicha distancia, pero es a partir de K = 95 cuando ya no varía.

Por otra parte, se han medido los tiempos de creación de los diferentes modelos para todos los valores de K desde 5 hasta 115, obteniendo como resultado que cuanto mayor es K, más tiempo se tarda en entrenar dicho modelo, tal y como se aprecia en la figura 4.3. La tendencia es creciente y lineal, ya que puede verse que no hay signos de estabilización temporal.

Conviene destacar que **el tiempo necesario para entrenar un modelo no tiene relación con lo bueno o malo que sea un valor de K**. El tiempo de cómputo de un modelo solo influye en el tiempo de entrenamiento, pero no en la eficiencia de dicho modelo. Una vez elegido el valor de K, se crea el modelo con dicho parámetro. Es posible serializar el modelo para que no sea necesario entrenarlo cada vez que se ejecuta el programa.

K	Tiempo (seg.) con 8 cores	Tiempo (seg.) con 1 core
5	283	1061
15	344	1410
25	434	1651
35	502	1941
45	616	2273
55	689	2543
65	798	2895
75	844	3185
85	900	3512
95	957	3799
105	1050	4133
115	1123	4450

Cuadro 4.1: Tiempo de cálculo de los modelos en función de K y los cores utilizados

El tiempo de cálculo de los modelos se ha medido utilizando un solo core y ocho al mismo tiempo sobre el 70 % de los 4.9 millones de registros que forman el conjunto de datos. Tal y como se aprecia en la figura 4.3, en la que están graficados los resultados de la tabla 4.1, a medida que crece el valor de K, más capacidad de cómputo se requiere y por lo tanto, más tiempo se tarda. Se puede decir que cuantos más cores y más máquinas se utilicen, mejor será el rendimiento.

```
1 clusters.save('model')
```

Fragmento de código 4.6: Serialización de un modelo

Productor Apache Kafka

El productor Apache Kafka actuará como generador de datos que deben ser analizados por el consumidor. El código será igualmente desarrollado en Python y se ha utilizado la librería Python-Kafka [24].

Los datos que envía este productor deben tener el mismo formato que los datos que han sido utilizados para entrenar el modelo.

El código es realmente sencillo, ya que unicamente se trata de ver cómo enviar datos desde el productor.

```
1 from kafka import Producer, KafkaClient
2
3 kafka = KafkaClient('IP_ADDRESS:PORT')
4 producer = Producer(kafka)
5 producer.send_messages('my-topic', dataToSend)
```

Fragmento de código 4.7: Productor Apache Kafka desarrollado en Python

Este código crear un cliente Apache Kafka con la clase **KafkaClient()** y un productor con la clase **Producer()**. Tras esto, con el método **send_messages()** se envía un mensaje con los datos a un determinado *topic*.

Al tratarse de una simulación, lo datos que se van a enviar ya están preprocesados y formateados correctamente.

Consumidor Apache Kafka

El consumidor Apache Kafka también estará desarrollado en Python y su función es la de recibir los datos del productor para más tarde realizar una predicción sobre los mismos. El consumidor hará uso del modelo entrenado y serializado previamente con el algoritmo KMeans para poder hacer las predicciones. El caso de uso aplicado en este apartado es el propuesto en la sección 3.2.2.

En la primera línea se carga el modelo y en la segunda se crea una instancia del consumidor, que recibe los parámetros necesarios para leer mensajes de un determinado *topic* y una determinada URL. Una vez configurada la conexión con el productor, el consumidor se pone a la espera para recibir nuevos mensajes, los cuales pasarán a ser un RDD que se introducirá como parámetro a la función de predicción.

```
1 savedModel = KMeans.load("model")
```

```
consumer = KafkaConsumer('my-topic',
           bootstrap_servers=['IP_ADDRESS:PORT'])
print("Waiting for messages...")
for message in consumer:
    data = sc.parallelize([message.value])
    testData = data.map(parseData)
    predictions = savedModel.predict(
                    testData.map(lambda x: x.features))
    print("Prediction: ")
    print(predictions.first())
```

Fragmento de código 4.8: Consumidor y predictor Apache Kafka en KMeans

La predicción se muestra por pantalla con la última línea de código y utilizando la función **first()** debido a que se extrae el valor de un RDD que únicamente contiene un resultado.

Lo óptimo sería tener muchos arrays de datos dentro del RDD del que se va a realizar la predicción para que pueda escalar bien la aplicación, pero para este caso de estudio funciona a la perfección.

4.2. Predicción de fallos online

La gestión de sistemas distribuidos y grandes redes de ordenadores es compleja. Cuando estos sistemas son críticos, es crucial asegurar su correcto comportamiento y rendimiento. No basta con asegurar su seguridad y minimizar el tiempo que el sistema no esté funcionando, sino que es necesario tener una actitud proactiva en este sentido, llamada **gestión de fallos proactiva**.

Esta disciplina tiene como objetivo predecir fallos un cierto tiempo antes de que ocurran para así poder tomar las medidas necesarias para evitarlos o minimizar los daños causados. Por lo general, este proceso está formado por cuatro pasos:

1. **Predicción de fallos online**: consiste en identificar posibles situaciones en las el sistema pueda fallar. Normalmente este punto se lleva a cabo mediante técnicas de data mining y machine learning.

2. **Diagnóstico**: una vez se sabe que va a producirse un fallo, hay que saber dónde y por qué.

3. **Planificación**: definición de las medidas a tomar para responder al futuro fallo del sistema.

4. **Reacción**: ejecutar las medidas propuestas en el paso anterior.

Este caso de estudio se centra en el primer paso, es decir, en predecir cuándo hay un fallo o no.

4.2.1. Conjunto de datos

El conjunto de datos de que se va a disponer para este caso consiste en un conjunto de eventos que ocurren en una infraestructura de red. Estos eventos pueden ser alertas de un disco lleno al 90 %, memoria RAM de un dispositivo llena o consumo de CPU más alto de 95 % en un determinado equipo de la red en cuestión.

La predicción de fallos se realizará a cinco minutos vista, es decir, el objetivo es poder avisar de que algo va a fallar con 5 minutos de antelación para poder diagnosticar el fallo, planificar las medidas y reaccionar ante dicho fallo. Para ello, los datos que se van a utilizar para realizar una predicción serán los eventos que han ocurrido en una ventana de cinco minutos, es decir, se tendrán en cuenta todos los eventos que han ocurrido dentro de ese intervalo temporal.

Sabiendo los eventos que pueden ocurrir, se preprocesan para dejar un array de valores *true* o *false* para saber si un determinado evento ha ocurrido dentro de la ventana temporal de cinco minutos. Finalmente se utiliza la última columna para saber si ha ocurrido un fallo en el sistema o no, que será usada para entrenar el modelo.

El archivo de datos está formado por 10661 líneas y 88 columnas, que se corresponden con los tipos de eventos. Tal y como se ha mencionado, los valores serán *true* o *false* en función de si un evento ha ocurrido o no, por lo que tendrán el aspecto siguiente:

- true, false, true, true, false, false, false, ...
- false, true, true, false, true, false, false, ...

4.2.2. Algoritmo utilizado

En este caso no se ha utilizado KMeans, sino un algoritmo de clasificación. Debido a la gran cantidad de variables que contiene el conjunto de datos, es conveniente probar el algoritmo Random Forests [21], el cual mejora la precisión en la clasificación mediante la incorporación de aleatoriedad en la construcción de un clasificador, la cual puede introducirse tanto en la construcción de los árboles como en los datos de entrenamiento.

4.2.3. Implementación

En esta parte del caso de estudio se va a explicar el proceso seguido para la implementación del código que soluciona este problema con Apache Spark.

Entrenamiento del modelo

Una buena aproximación para solucionar el problema de la predicción de fallos online sería haciendo uso del algoritmo de regresion logística, pero tal y como se menciona en el apartado anterior, se ha creído conveniente probar el algoritmo Random Forests para ver si se comporta bien ante estos datos. Debido a esto, ya no se va a utilizar la librería de clustering, sino la de árboles de MLLib.

```
1 from pyspark.mllib.tree import RandomForest
```

Fragmento de código 4.9: Librería Random Forests de PySpark

Entrenar un modelo con Spark y MLLib es muy parecido en todos los algoritmos. Lo primero que hay que hacer es, por supuesto, cargar los datos con la función **textFile** de Spark. Tras esto, se dividen los datos en entrenamiento y test. Los primeros serán para entrenar el modelo y los segundos para probarlo.

```
1 rawData = sc.textFile('file.data')
2 data = rawData.map(parseData)
3 (trainingData, testData) = data.randomSplit([0.7, 0.3])
```

Fragmento de código 4.10: Carga de datos y separación en entrenamiento y test

Tal y como se aprecia en la línea 2, se realiza un mapeado de los datos con la función **parseData()**. Esto se debe a que el método de Random Forests que entrena un modelo no acepta los datos como una cadena de caracteres con parámetros separados por comas, sino que necesita una entrada del tipo **LabeledPoint**. El formato de este tipo de datos es el siguiente: si hay un registro con cuatro columnas, siendo las tres primeras variables de los datos y la última el *target* en este formato: `1, 2, 3, 4`, un *labeled point* forma un vector con las tres primeras columnas y le pone la etiqueta de la cuarta columna: `(4, [1, 2, 3])`. Con la función **parseData()** se realiza la

conversión a este tipo de dato para que sea compatible con Random Forest.

```
1 def parseData(line):
2     values = [float(s) for s in line.split(",")]
3     label = values[-1]
4     featuresVector = Vectors.dense(values[0:-1])
5     return LabeledPoint(label, featuresVector)
```

Fragmento de código 4.11: Función para crear LabeledPoints

Una vez se tienen los datos en el formato correcto, se llama a la función **RandomForest.trainClassifier()**, que espera varios parámetros:

- Datos: los datos de entrenamiento en formato LabeledPoint.
- Número de clases: las diferentes clases que existen en el *target*. En este caso solo hay *true* o *false*, es decir, 2.
- Número de árboles.
- Profundidad.
- Otros parámetros relacionados con el algoritmo.

```
1 model = RandomForest.trainClassifier(
2                       trainingData,
3                       numClasses=10,
4                       categoricalFeaturesInfo={},
5                       numTrees=5,
6                       featureSubsetStrategy="auto",
7                       impurity='gini',
8                       maxDepth=5,
9                       maxBins=32)
```

Fragmento de código 4.12: Entrenamiento de un modelo Random Forests

Una vez creado el modelo, es posible serializarlo y almacenarlo en disco, igual que en el caso de estudio anterior.

En la figura 4.4 se puede apreciar que no hay una gran diferencia de rendimiento en el cálculo de los modelos utilizando uno u ocho cores de la

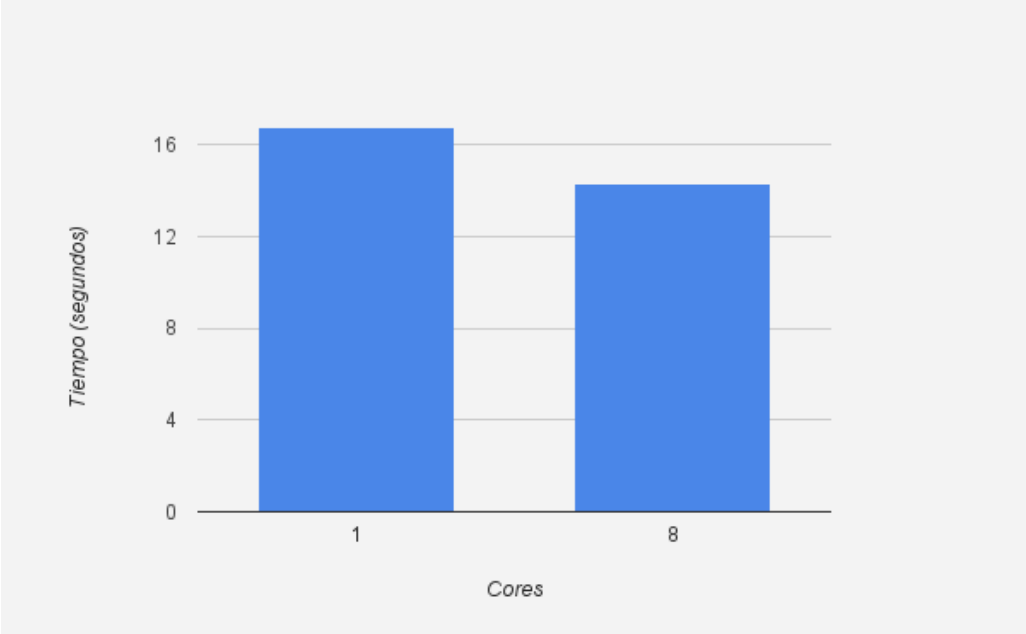


Figura 4.4: Tiempo de cálculo de los modelos en función de los cores utilizados.

Cores	Tiempo (segundos)
1	16,747
8	14,344

Cuadro 4.2: Tiempo de cálculo de los modelos en función de los cores utilizados

máquina debido a que la carga de trabajo no es tan grande como para utilizar todos los recursos del sistema. Estos cálculos se han realizado entrenando el 70 % del conjunto de datos, que son unas 7000 líneas. En el caso de estudio anterior se utilizaba el 70 % de casi 5 millones de líneas y de ahí la diferencia.

Productor Apache Kafka

El productor Apache kafka de este caso de estudio es exactamente el mismo que el del caso de kMeans. La única diferencia que podría haber es el *topic* que se utilice, pero los datos van a ser mandados en el mismo formato,

es decir, como una cadena de caracteres separados por comas.

En este caso, sería conveniente utilizar también Spark Streaming por las funcionalidades que ofrece con su implementación de ventanas temporales, pero no se ha utilizado debido a que no existe un streaming de datos real y los datos ya están preprocesados. En caso de no tener datos preprocesados, sí sería necesario utilizar esta tecnología.

Tal y como se ha comentado en el apartado de análisis de datos de este mismo caso de estudio, los eventos producidos en un sistema, como por ejemplo que el consumo de CPU sea mayor de un 90 % o que ya no quede memoria disponible, deben ser analizados en función de datos anteriores. Con Spark Streaming llegarían datos en ventanas de 5 minutos y se formaría el *array* de datos que se va a utilizar en este ejemplo. Debido a que los datos ya están procesados tanto para entrenar el modelo como para realizar la predicción, no se utilizará Spark Streaming, sino Apache Kafka para enviar dichos datos al predictor.

Consumidor Apache Kafka

El consumidor en este caso también es parecido al del caso anterior. La única diferencia notable es el tipo de modelo que se va a utilizar para realizar las predicciones, que es el Random Forests, pero la manera en que va a ser leído es exactamente la misma que con KMeans. Podría decirse que cualquier modelo serializado de la librería MLLib se lee igual.

Al comienzo del código y tras haber importado el modelo previamente entrenado, se crea el consumidor Apache Kafka, que recibirá los datos y creará un RDD con ellos para poder hacer las predicciones.

```
1 savedModel = RandomForests.load("model")
2 consumer = KafkaConsumer('my-topic',
```

```
3             bootstrap_servers=['IP_ADDRESS:PORT'])
4 print("Waiting for messages...")
5 for message in consumer:
6     data = sc.parallelize([message.value])
7     testData = data.map(parseData)
8     predictions = savedModel.predict(
9                       testData.map(lambda x: x.features))
10    print("Prediction: ")
11    print(predictions.first())
```

Fragmento de código 4.13: Consumidor y predictor Apache Kafka en Random Forests

La predicción sobre dichos datos se realiza con la función **predict()** y únicamente recibe como parámetro los vectores que forman parte del LabeledPoint obtenido del RDD con la función **parseData()**.

Tal y como se aprecia en los fragmentos de código anteriores, el formato de los datos utilizados para crear el modelo y predecir valores es siempre el mismo: LabeledPoint. Por otra parte, una vez más es posible afirmar que haciendo uso de Spark Streaming la ejecución sería más eficiente, debido a que habría más información en los RDD y podría paralelizarse el proceso de predicción obteniendo una velocidad mayor.

4.3. Análisis de sentimientos en Twitter

El análisis de sentimientos es un proceso en el cual se intenta predecir el tipo de sentimiento que una persona pudo sentir o intentó expresar al momento de escribir un texto, en este caso mediante una publicación en la red social Twitter. El objetivo de este caso de estudio es el de valorar y determinar si el usuario de Twitter ha publicado un mensaje positivo, negativo o neutro sobre un tema determinado.

Este caso de estudio ha sido desarrollado en **Scala** con el objetivo de mostrar las diferencias con los anteriores casos de estudio, desarrollados en

Python.

4.3.1. Conjunto de datos

El conjunto de datos que se va a utilizar está formado por mensajes de Twitter, los cuales son obtenidos directamente desde la red social a través de Spark Streaming.

Para poder acceder a dichos mensajes es necesario crear una aplicación desde la consola de desarrolladores de Twitter con el fin de obtener cuatro parámetros:

- Consumer key.
- Consumer secret.
- Access token.
- Access token secret.

Para ver cómo se crea una aplicación de Twitter se puede consultar el apéndice 2, en el que se explica el proceso.

Una vez obtenidas las claves, se van a poner en un archivo de configuración y la aplicación automáticamente leerá ese archivo para poder acceder a Twitter y descargar los tweets. El problema de Twitter es que proporciona muy pocos tweets por por cada llamada a su API, cosa que podría solucionarse haciendo muchas llamadas, pero también están muy limitadas dichas llamadas [25] y si se realizan demasiadas, se cortará la comunicación de la aplicación con Twitter. En cinco segundos se van a recibir 10 tweets. No es una cantidad excesiva de mensajes, ni mucho menos, pero aunque no sea muy grande podemos ver cómo funciona Spark Streaming y su sistema de ventanas.

4.3.2. Implementación

Obtener datos

Una vez creada la aplicación y se tengan los cuatro parámetros necesarios para obtener los mensajes de Twitter, se van a introducir en el fichero de configuración de Twitter4J [23]: **twitter4j.properties**.

```
1 oauth.consumerKey = XXXXXXXXXXXXXXXXXXXXXXXXX
2 oauth.consumerSecret = XXXXXXXXXXXXXXXXXXXXXXXXX
3 oauth.accessToken = XXXXXXXXXXXXXXXXXXXXXXXXX
4 oauth.accessTokenSecret = XXXXXXXXXXXXXXXXXXXXXXXXX
```

Fragmento de código 4.14: Configuración de la API de Twitter

Una vez configurado el archivo de propiedades ya es posible crear un stream de Twitter haciendo uso de la librería **TwitterUtils** de Apache Spark Streaming. Lo que hace esta librería es obtener los tweets dentro de una ventana temporal que en este caso es de 5 segundos, es decir, cada 5 segundos nos van a llegar mensajes. Al ejecutar el código se ve que llegan 10 mensajes en esa ventana, por lo que se podría decir que llegan a una media de dos mensajes por cada segundo. Como ya se ha mencionado anteriormente, es muy poco dado que es una API para proyectos experimentales y no para poner en producción.

La API tiene limitaciones en cuanto a las llamadas que se hacen y, se ha elegido una ventana de cinco segundos debido a que el límite es de 180 peticiones en 15 minutos, que equivale a una petición cada cinco segundos.

```
1 val sc = new SparkContext(conf)
2 val ssc = new StreamingContext(sc, Seconds(5))
3 val filters = new Array[String](1)
4 filters(0) = "greece"
5 val twitterStream = TwitterUtils.createStream(ssc,
6                                                 None,
7                                                 filters)
```

Fragmento de código 4.15: Creación de un streaming de Twitter

Es posible añadir filtros al streaming de Twitter mediante un array, que equivale a la búsqueda que se puede realizar a través de la propia web. En el código anterior se ha buscado la palabra "greece", para que de resultados relacionados con dicho filtro, obteniendo 10 mensajes cada cinco segundos. Los dos primeros recibidos han sido los siguientes:

```
    Time: 1436353630000 ms
---------------------
(6187380632,RT @IvanaKottasova: Tried to buy a train ticket in
Athens. Ït's free", I am told. "The banks are closed."#Greece)
(6187380678,Greece has the opportunity to repudiate a lot of
inconvenient foreign debt.)
```

Análisis

Una vez se han recibido los datos se puede pasar a realizar su análisis, pero antes deben importarse las palabras positivas y negativas [19], las cuales juegan un papel fundamental en este caso de estudio.

```
1 val positive_words =  sc.textFile("positivas.txt")
2       .filter(line => !line.isEmpty())
3       .collect()
4       .toSet
5
6 val negative_words = sc.textFile("negativas.txt")
7       .filter(line => !line.isEmpty())
8       .collect()
9       .toSet
```

El proceso de análisis de un mensaje es el siguiente:

1. Eliminar los números y caracteres extraños del mensaje.

2. Crear un array de las palabras que quedan en el mensaje.

3. Contar palabras y puntuar el mensaje.

Para eliminar los números y caracteres extraños del mensaje se define la función **clean()**, que recibe como parámetro un texto y devuelve una lista de palabras.

```
def clean(tweetText: String): List[String] =
    tweetText.split(" ").
    map(_.toLowerCase).
    filter(_.matches("[a-z]+")).toList
```

Fragmento de código 4.16: Función de limpieza de un mensaje de Twitter

La función **rate()** recibe como parámetro una única palabra y comprueba si dicha palabra está en la lista de palabras positivas, negativas o en ninguna. En caso de ser una palabra positiva suma un punto, en caso de ser negativa, resta un punto y en el resto de casos, no se suma nada. Esta función es llamada por **rateWordList()**, que recibe como parámetro un array de palabras y ejecuta para cada una de ellas la función anterior.

```
def rate(word: String): Int =
    if (positive_words.contains(word))       1
    else if (negative_words.contains(word)) -1
    else                                     0

def rateWordList(words: List[String]): Int =
    words.foldRight(0) {
        (word, score) =>
            score + rate(word)
    }
```

Fragmento de código 4.17: Funciones para puntuar un mensaje

Una vez se han definido las funciones, van a aplicarse a todos los mensajes recibidos por streaming con la función **map()**.

```
twitterStream.
   map{tweet => (tweet.getId(), tweet.getText())}.
   map{case (id, text) => (id, clean(text))}.
   map{case (id, words) => (id, rateWordList(words))}.
   print()
```

Como resultado final, se obtiene el identificador del mensaje y su puntuación. También es posible imprimir el mensaje y su puntuación, pero para comprimir al máximo la salida por pantalla, se ha optado por mostrar únicamente el identificador:

```
Time: 1436361275000 ms
---------------------
(618770130890133505,1)
(618770131334918144,0)
(618770132014362624,0)
(618770132144377856,0)
(618770133599825920,1)
(618770135097192448,0)
(618770136187727873,-1)
(618770136149790720,0)
(618770137676517376,1)
(618770138267914240,-2)
```

4.4. Resumen

En este capítulo se han realizado tres casos de estudio: detección de anomalías en redes, predicción de fallos online y análisis de sentimientos en Twitter. Tras implementar y resolver los tres problemas, se ha demostrado que la arquitectura propuesta es válida para los tres casos.

Tal y como se ha podido comprobar, la paralelización de las tareas de Apache Spark hace que el programa sea más eficiente, aunque no siempre hay una gran diferencia y depende del conjunto de datos, el algoritmo y el

caso de estudio. En los casos de estudio propuestos se han utilizado diferentes algoritmos y métodos de recepción de datos de fuentes externas. Al conseguir resolver los tres problemas planteados, se puede afirmar que Apache Spark es válido para realizar tareas de machine learning gracias a su librería MLLib y la integración con librerías de streaming, como Spark Streaming.

Por otra parte, el uso de las ventanas temporales de Spark Streaming es una ventaja por la facilidades que proporciona un DStream al contener varios RDD. El uso de cualquier algoritmo dependerá del caso de estudio, pero tal y como se acaba de mencionar, Spark proporciona facilidades para tareas de machine learning al contener funciones específicas para entrenar modelos, además de múltiples algoritmos que se aplicarán a los RDD.

El principal problema encontrado ha sido el tiempo de respuesta para dar el resultado de una predicción en los dos primeros casos de estudio cuando se han mandado miles de datos por segundo. Esto es debido a que en la simulación se ha utilizado una única máquina, perdiendo así capacidad de cómputo en comparación con un cluster de varios servidores. La solución a esto es utilizar más de un servidor aplicando la arquitectura propuesta para poder escalar sin problema y no generar un cuello de botella en las predicciones con un único ordenador.

La mayor complejidad de los casos de estudio reside en la implementación de la arquitectura en el sentido de conectar cada bloque. Las fuentes de datos externas tienen que enviar los datos en el formato correcto y Spark necesita recibirlos en ese formato. Tras recibirlos, hay que preprocesarlos para el tipo de algoritmo que se vaya a utilizar, como los LabeledPoint o vectores. El tener que convertir datos de un formato a otro incrementa la carga de trabajo en el sistema, pero no es donde se genera el cuello de botella. Como ya se ha mencionado, para reducir el tiempo de respuesta de Spark ante nuevos datos,

lo óptimo es disponer de una alta capacidad de procesamiento y recursos hardware suficientes en función del problema.

Capítulo 5

Conclusiones y líneas futuras

En el trabajo presentado se ha realizado un análisis de las arquitecturas de procesado de streaming Big Data existentes.

En el primer capítulo se han introducido los conceptos básicos del Big Data, así como de la ciencia de datos y analítica predictiva y descriptiva. A continuación, se han expuesto con detalle las plataformas existentes para tratamiento y análisis de grandes cantidades de datos. Dicho análisis se ha dividido en varias categorías: tratamiento de datos, tecnologías streaming y lenguajes de programación.

Apache Hadoop MapReduce fue en su día una herramienta clave en el tratamiento masivo de datos, pero con el desarrollo de Apache Spark, se queda muy por detrás en cuanto a rendimiento y capacidad de procesamiento, ya que Apache Spark es capaz de gestionar muchos más datos en mucho menos tiempo y con menos máquinas, ahorrando así mucho coste a la hora de desplegar la arquitectura propuesta en el capítulo tercero.

Por otra parte, la elección de una tecnología de streaming determinada dependerá del caso de estudio, igual que los lenguajes de programación. La mejor opción para desarrollar código programando sobre Spark es Scala, sien-

do Java muy similar. Sin embargo, la ventaja de Python es la simplicidad con la que permite escribir líneas de código. El único problema de Python es que su API de programación en Spark no está a la altura de las de Scala o Java.

En el tercer capítulo se han establecido los requisitos que debe tener un sistema de procesado de Big Data con streaming y también se ha propuesto una arquitectura como posible solución a un problema que tenga los requisitos establecidos en este capítulo

Por otra parte, en el cuarto capítulo se han planteado tres casos de estudio diferentes y se ha comprobado que Apache Spark es totalmente válido para tareas de machine learning y análisis de datos recibidos por streaming.

Los objetivos propuestos han sido cumplidos, ya que se ha realizado el análisis de las arquitecturas de streaming Big Data existentes, ejecutando además varios casos de estudio que ponen en práctica las herramientas estudiadas a lo largo del trabajo.

Como trabajo futuro se propone la creación de visualizadores de resultados con entorno gráfico, ya que la salida por consola no es suficiente para entornos de producción. Además, sería interesante comprobar el rendimiento de los experimentos realizados en este trabajo haciendo uso de un cluster de Spark formado por varias máquinas.

Apache Spark es un proyecto que crece día a día y está en constante desarrollo. Gracias a esto, como trabajo futuro también se pueden probar nuevos algoritmos en las distintas APIs de programación.

Bibliografía

[1] Apache Spark. http://spark.apache.com/

[2] Apache Kafka. http://kafka.apache.com/

[3] Shvachko, Konstantin, et al. "The hadoop distributed file system."Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on. IEEE, 2010.

[4] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters.Çommunications of the ACM 51.1 (2008): 107-113.

[5] Apache Zookeeper. https://zookeeper.apache.org/

[6] Zaharia, Matei, et al. "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters."Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing. USENIX Association, 2012.

[7] Zaharia, Matei, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing."Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.

[8] Apache Oozie. http://oozie.apache.org/

[9] Azkaban. https://azkaban.github.io/

[10] API Python Apache Spark.
https://spark.apache.org/docs/latest/api/python/pyspark.html

[11] API Scala Apache Spark.
https://spark.apache.org/docs/latest/api/scala/index.html

[12] API Java Apache Spark.
https://spark.apache.org/docs/latest/api/java/index.html

[13] Sort Benchmark. http://sortbenchmark.org/

[14] Scala. http://www.scala-lang.org/

[15] Apache Storm. https://storm.apache.org/

[16] C. Kruegel, G. Vigna. "Anomaly Detection of Web-based Attacks". Proceedings of the 10th ACM conference on Computer and communications security. Pages 251 - 261. ISBN:1-58113-738-9.

[17] Conjunto de datos detección de anomalías. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

[18] Lane, Terran, Carla E. Brodley. .An application of machine learning to anomaly detection."Proceedings of the 20th National Information Systems Security Conference. Vol. 377. Baltimore, USA, 1997.

[19] Diccionario de palabras positivas y negativas. http://www.cs.uic.edu/ liub/FBS/sentiment-analysis.html#datasets

[20] Likas, Aristidis, Nikos Vlassis, Jakob J. Verbeek. "The global k-means clustering algorithm."Pattern recognition 36.2 (2003): 451-461.

[21] Breiman, Leo. Random forests."Machine learning 45.1 (2001): 5-32.

[22] Consola de Twitter para desarrolladores.

https://apps.twitter.com/

[23] Librería Twitter4J. http://twitter4j.org/en/index.html

[24] Librería Kafka-Python.

https://github.com/mumrah/kafka-python.git

[25] API pública de Twitter. https://dev.twitter.com/rest/public

Apéndices

Apéndice A

Creación de un cluster

Para crear un cluster podemos hacer uso de servidores reales o máquinas virtuales en local. Debido a que en este proyecto no se han tenido servidores disponibles para este fin, se realizará con máquinas virtuales, por lo que se utilizará:

- Herramienta de virtualización: VMware o VirtualBox.
- Sistema operativo: Ubuntu 14.04.

Requisitos

Crear un cluster de servidores Apache Spark es muy parecido a como se crearía un único servidor de Spark. Lo único que hay que hacer para añadir máquinas es poner sus direcciones IP en un archivo de configuración. Sin embargo, hay que tener un determinado software instalado:

- Java: se puede utilizar tanto la versión 7 como la 8.
- Scala: se instalará la última versión.

- Acceso remoto por SSH entre máquinas.
- Apache Spark pre-compilado para Hadoop.

Máquinas virtuales

Todas las máquinas virtuales deben tener software en común, que es el que se ha mencionado antes: Java, Scala y Spark. Al tener Ubuntu recién instalado, habrá que configurar el sistema para tener el software disponible. Sin embargo, no es deseable configurarlo todo cada vez para cada máquina, por lo que se hará uso de la función de clonado de VMware para configurar un único servidor y luego clonarlo.

Paso 1

El primer paso es crear la máquina Ubuntu, que se llamará **spark-master**, para saber en qué máquina se está trabajando. Se ha dejado toda la configuración por defecto excepto la memoria RAM, que tendrá 2GB.

Cuando ya se haya instalado el sistema operativo por completo, veremos algo parecido a lo siguiente:

Nota: los pasos que vienen a continuación se harán en la máquina **ubuntu-spark-master**.

Paso 2

En este paso se instalará Java. Vale tanto la versión openJDK como la de Oracle. En este caso se va a instalar Java 8 de Oracle. Para ello, hay que introducir los comandos que se mostrarán a continuación.

Antes de nada, conviene actualizar el sistema:

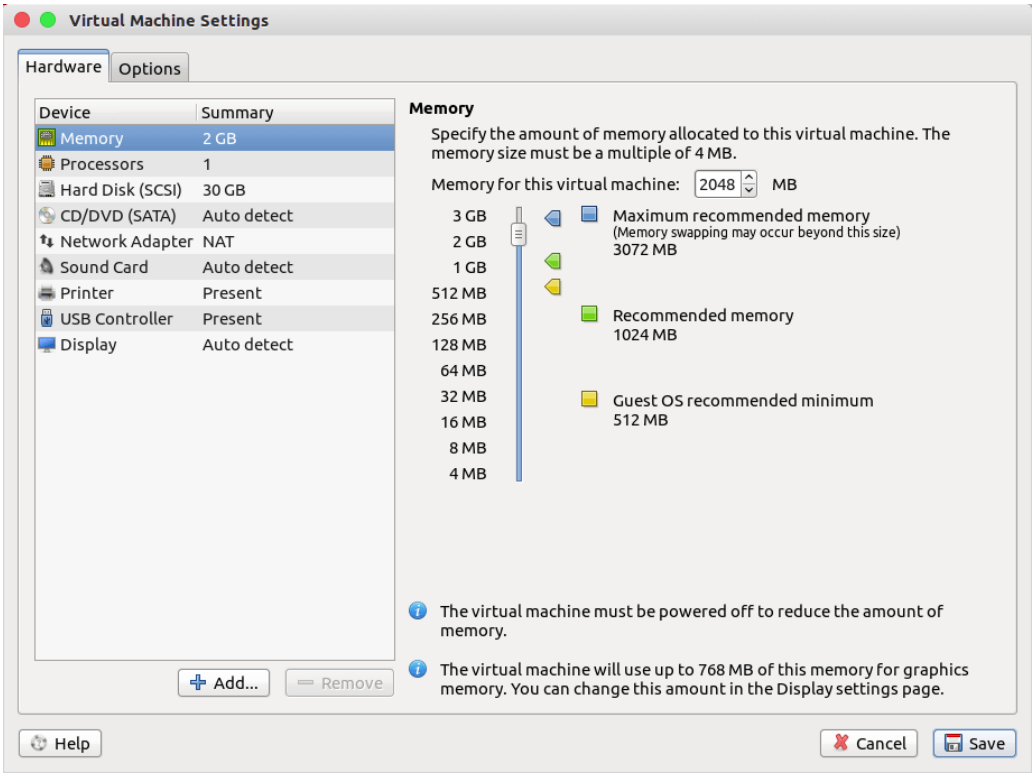


Figura A.1: Configuración de la máquina virtual.

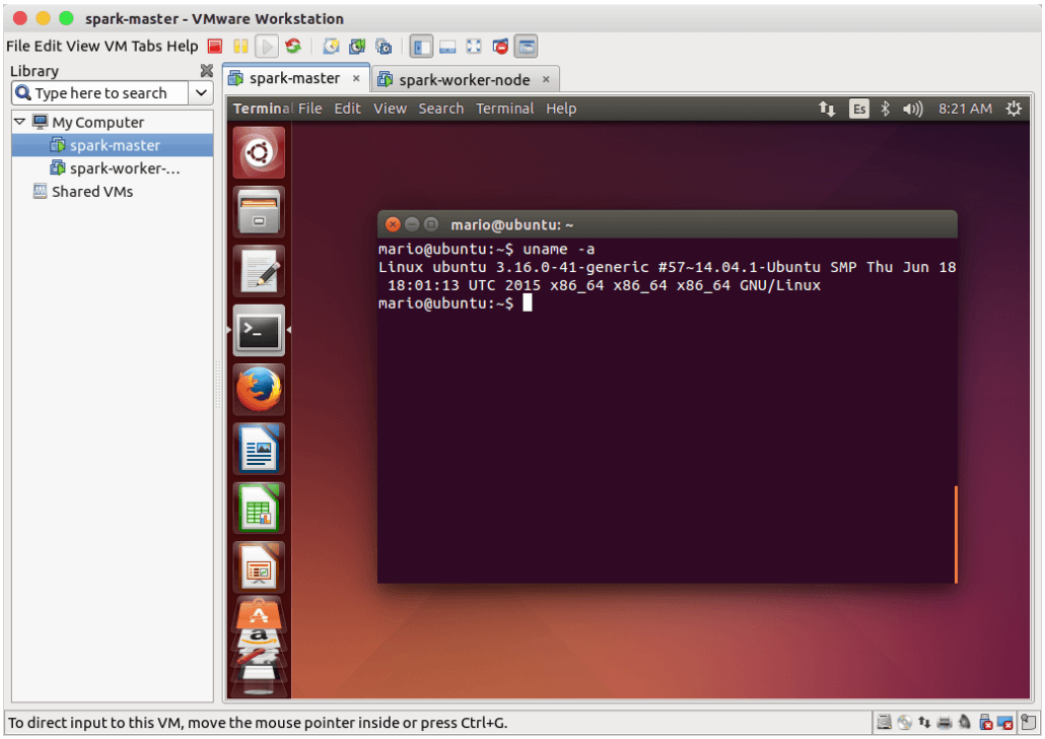


Figura A.2: Sistema operativo.

```
$ sudo apt-get update
$ sudo apt-get upgrade -y
```

Tras el paso anterior, se procede a instalar Java:

```
$ sudo apt-add-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

Paso 3

Este tercer paso va a consistir en instalar Scala [14].

Paso 4

Para comenzar con el cuarto paso hay que apagar la máquina virtual, ya que no se pueden clonar máquinas si la de origen está encendida. Una vez esté apagada, click con el botón derecho en el nombre de la máquina y dentro de "Manage", seleccionar "clonar". El tipo de clonación va a ser "linked" o "enlazada" y el nombre de la nueva máquina virtual: **ubuntu-spark-worker-node**.

Paso 5

En este paso se van a obtener las direcciones IP de ambas máquinas porque van a ser necesarias durante el proceso. De esta manera, se dejan apuntadas para el tutorial y no hay que estar siempre comprobándolas con `ifconfig`:

- **ubuntu-spark-master**: 192.168.107.128
- **ubuntu-spark-worker-node**: 192.168.107.129

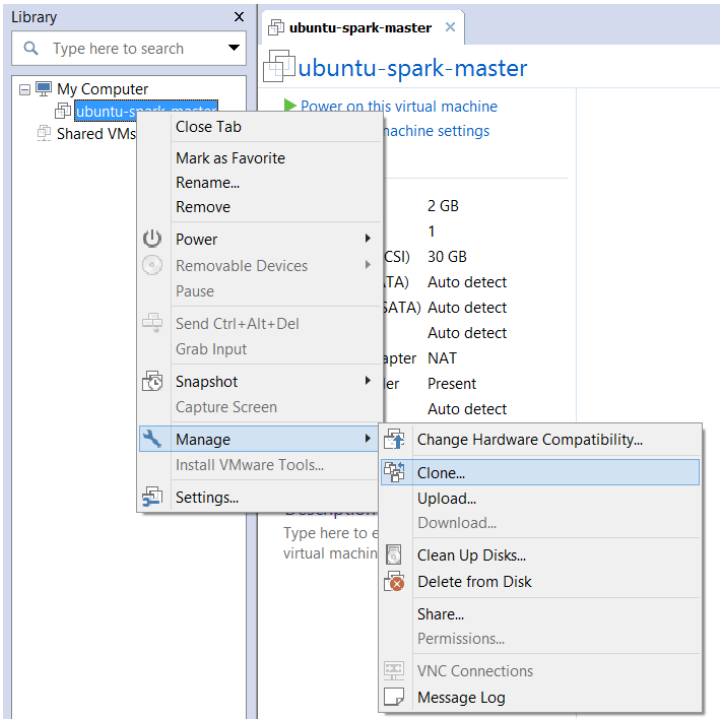


Figura A.3: Clonación de máquinas virtuales.

Paso 6

Apache Spark necesita acceder por SSH a las máquinas, pero sin contraseña, por lo que habrá que crear una clave para permitirle establecer la conexión.

En la máquina **ubuntu-spark-worker-node** se introduce este comando:

```
$ sudo apt-get install openssh-server
```

En la máquina **ubuntu-spark-master** generamos una clave RSA para el acceso remoto a los workers:

```
$ ssh-keygen
```

Para tener acceso a los workers sin contraseña, la clave RSA generada en el master debe ser copiada en cada uno de los workers. Por lo tanto, en la

máquina **ubuntu-spark-master** se ejecuta el siguiente comando:

```
$ ssh-copy-id -i ~/.ssh/id_rsa.pub \\
usuario_worker_1@IP.DEL.WORKER.1
```

Teniendo en cuenta lo anterior y sabiendo que únicamente hay un worker y su dirección IP es 192.168.107.129 con nombre de usuario *mario*, en **ubuntu-spark-master** se introduce el siguiente comando:

```
$ ssh-copy-id -i ~/.ssh/id_rsa.pub \\
mario@192.168.107.129
```

Paso 7

En este paso se va a descargar Apache Spark desde su página web. A día de hoy, la versión más reciente es la 1.3 y se va a descargar el paquete pre-compilado para Apache Hadoop 2.4 y posteriores. Al estar pre-compilado el proceso será más rápido.

La descarga de Apache Spark hay que realizarla en todas las máquinas, tanto master como workers.

Una vez descargado, se descomprime en la carpeta /home/usuario, quedando en este caso en la ruta /home/mario/spark-1.3-bin-hadoop2.4.

Para probar que todo funciona correctamente, dentro de la carpeta bin se ejecuta el comando `./spark-shell` para obtener como resultado una consola de Spark.

Paso 8

El octavo paso consiste en la configuración de Apache Spark, en la que será indicado quién es el nodo master y quiénes son los workers.

Configuración de los esclavos

Se abre la carpeta de Spark en **ubuntu-spark-master**. Dentro de la carpeta 'conf' hay un archivo llamado 'slaves.template', el cual debe ser renombrado a 'slaves'. Este archivo contiene las direcciones IP de los nodos esclavos (workers), por lo que se comentan todas las líneas con '#' y añadimos la IP de **ubuntu-spark-node-worker** para que finalmente quede así:

```
# localhost
192.168.107.129
```

A continuación, se copia este archivo en la misma ruta del **ubuntu-spark-node-worker**.

Configuración del entorno Apache Spark

Se abre la carpeta de Spark en **ubuntu-spark-master**. Dentro de la carpeta 'conf' hay un archivo llamado 'spark-env.sh.template', el cual se va a renombrar a 'spark-env.sh'. Hay que añadir las siguientes líneas de configuración:

- IP del nodo master:

  ```
  export SPARK_MASTER_IP = IP.UBUNTU.SPARK.MASTER
  ```

- Número de cores que se ejecutan en la máquina:

  ```
  export SPARK_WORKER_CORES=1
  ```

- Memoria total que un worker tiene disponible:

  ```
  export SPARK_WORKER_MEMORY=800m
  ```

- Número de procesos worker por cada nodo:

  ```
  export SPARK_WORKER_INSTANCES=2
  ```

De la misma manera que con el archivo 'slaves', se copia este archivo a

todos los nodos dentro de la misma ruta. En este caso a **ubuntu-spark-node-worker**.

Paso 9

En este último paso se hará una prueba de funcionamiento para ver que todo está configurado correctamente. Para ello, se ejecutarán algunas operaciones.

Paso 9.1: Probar la versión local

Dentro de la carpeta 'bin' se ejecuta el siguiente comando para calcular el número Pi con Spark:

```
$ ./run-example SparkPi
```

Si todo va bien, debe mostrar por pantalla el valor de Pi.

Paso 9.2: Probar el cluster

Dentro de la carpeta **sbin** existen varios ejecutables:

- Comando para iniciar el cluster: `./start-all.sh`
- Comando para detener el cluster: `./stop-all.sh`

Tras iniciar el cluster, se navega desde **spark-node-master** a la URL `localhost:8080` para ver el panel web, en el que se deberían encontrar dos nodos worker:

```
$ MASTER=spark://IP.UBUNTU.SPARK.MASTER:7077 \\
  ./run-example SparkPi
```

Como resultado, es posible observar en el panel web que se está ejecutando una aplicación en el cluster:

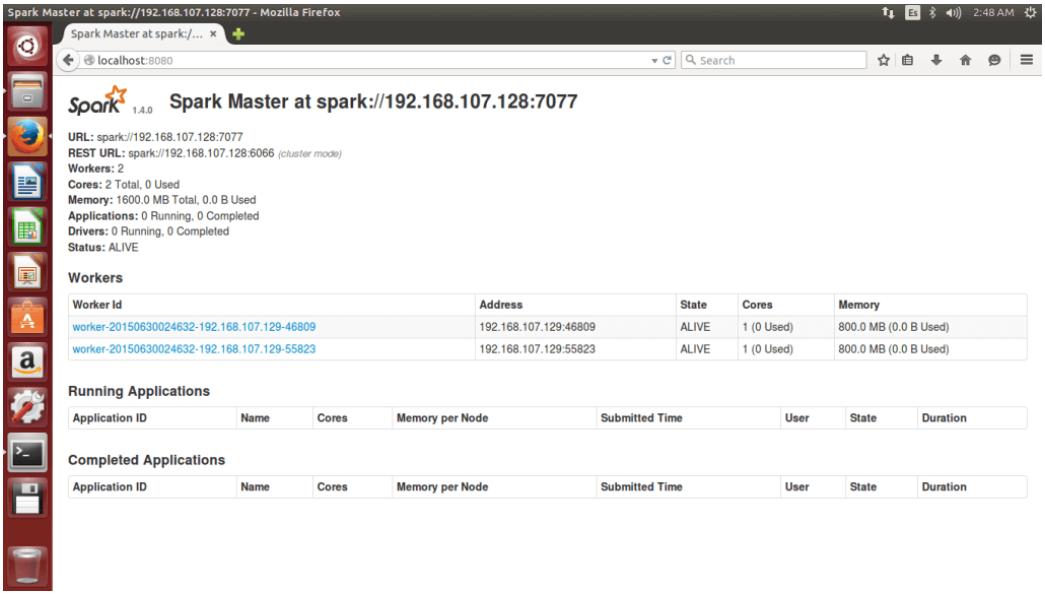


Figura A.4: Panel web del cluster de Apache Spark.

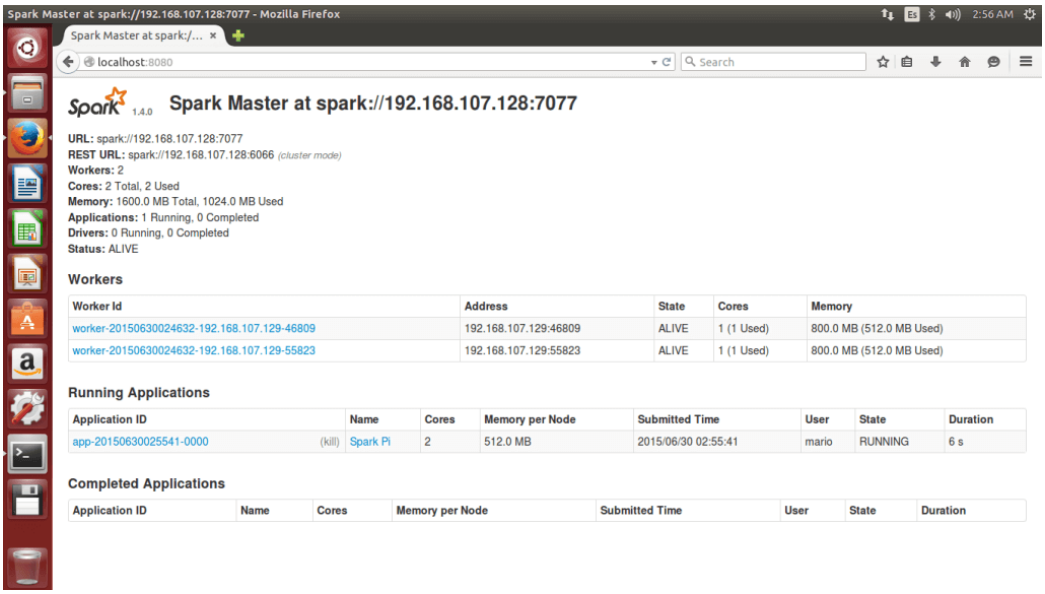


Figura A.5: Panel web del cluster de Apache Spark ejecutando una aplicación.

Si mientras se ejecuta el programa se hace uso del comando top en la máquina **ubuntu-spark-worker-node**, es posible observar que se están ejecutando dos procesos java que se corresponden con Apache Spark.

Apéndice B

Creación de una aplicación de Twitter

Para poder utilizar la API de Twitter es necesario crear una aplicación, cosa que se puede hacer en dos sencillos pasos. El objetivo de esto es obtener cuatro parámetros de dicha aplicación:

- Consumer key.
- Consumer secret.
- Access token.
- Access token secret.

Paso 1

Para crear la aplicación, hay que dirigirse a la página de Twitter para desarrolladores [22] y crear una nueva aplicación con los datos necesarios, que son los que aparecen en la figura B.1:

- Nombre de la aplicación.

- Descripción.
- Página web.

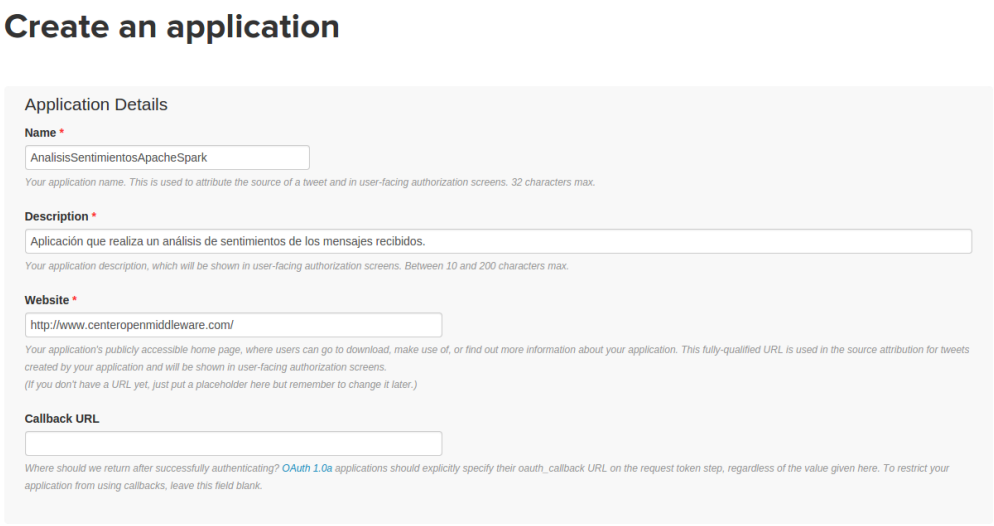


Figura B.1: Creación de una aplicación en Twitter.

Paso 2

Una vez creada la aplicación, se pueden ver las claves de acceso en la sección **Keys and Access Tokens**, tal y como se puede apreciar en la figura B.2. Sin embargo, están las claves de **consumer**, pero no de **access**, por lo que deben crearse, obteniendo como resultado lo que se muestra en la figura B.3.

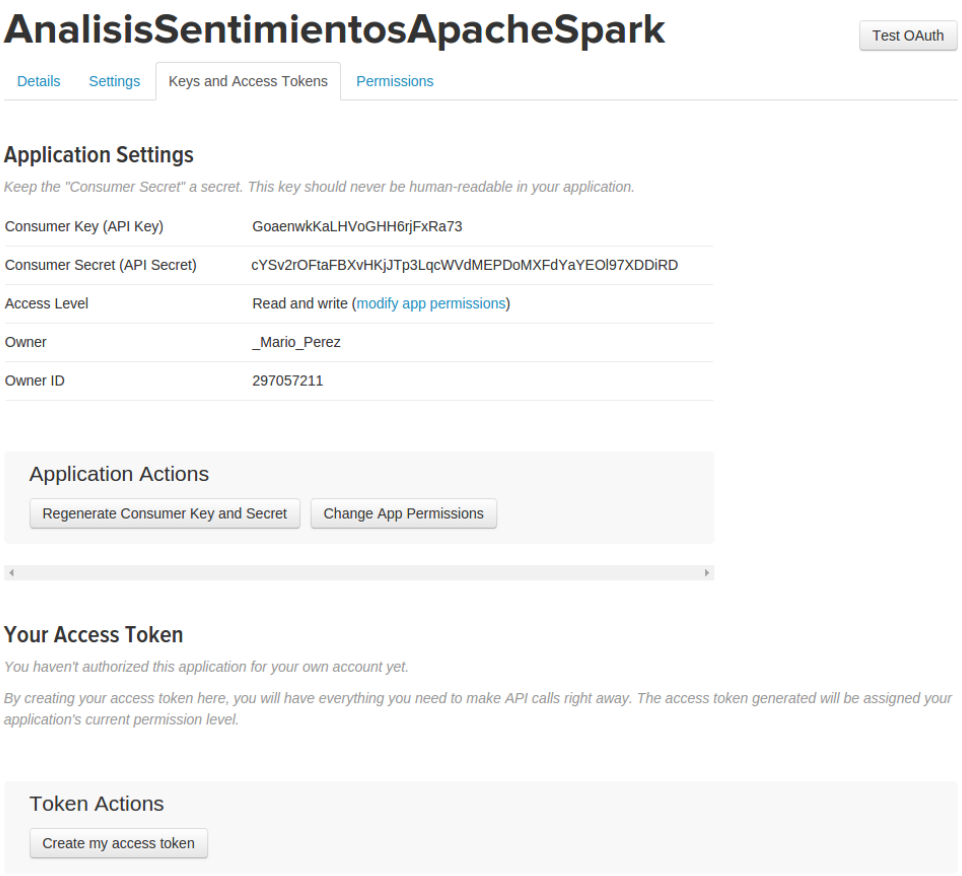


Figura B.2: Claves de la aplicación.

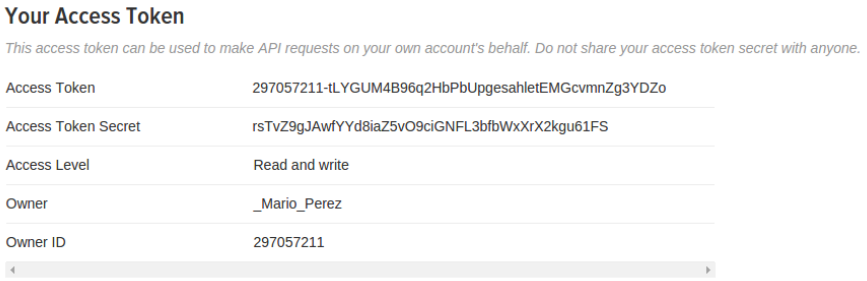


Figura B.3: Tokens de acceso.