UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Departament d'arquitectura de computadors

MULTIMEDIA BIG DATA COMPUTING FOR TREND DETECTION

Por:

Omar Iván Sulca Correa

Director:

Prof. Dr. Ruben Tous Liesa

MASTER THESIS

Màster en Enginyeria Informàtica

Facultat d'nformàtica de Barcelona (FIB)

Universitat Politecnica de Catalunya (UPC) – BarcelonaTech

Abril - 2015

Abstract

The Big data analysis has becoming increasingly more relevant for the enterprises because the efficient handling of information represents a unique competitive advantage, being its application so diverse as the nature of the data. Ejm. Fraud detection, advertising strategies, web traffic monitoring, etc.

Apache Spark is a engine for large-scale data processing, intended to be a drop in replacement for Hadoop MapReduce providing the benefit of improved performance; the main goal of this project is proof the capabilities of this system, through the development and implementation of a distributed pipeline for processing and indexing at high speed and real-time multimedia data streams generated by social networks and detect trends in these, using for this purpose the Spark related projects and libraries: Spark Streaming and Spark MLlib.

To verify the effectiveness of the algorithm, different benchmarks (with different configurations) will be performed, these results will be analyzed.

Agradecimientos

Me gustaria darle un agradecimiento especial a mi director de tesis Dr. Rubén Tous, por permitirme participant en esta tesis y por su paciencia, soporte, guia y aliento durante su elaboración.

A mis amigos Andrea, Albert, Jorge, Juan y Leonardo, por su ayuda incondicional y útiles consejos a través de estos meses

Y por supuesto, al gobierno peruano por darle esta oportunidad única.

Dedicado a mi familia. A Ana, mi madre, por sus sabios consejos y su infinito cariño; a Cayetano, mi padre, un incansable motivador y mi aspiración como persona y profesional, y a Jimmy, Kevin y Harold, a mis hermanos, un parte importante en todo lo que hago.

Tabla de contenido

Tabla de ilustraciones

1. Introducción

1.1. Contexto

La relevancia de los datos de la *social media* (imágenes, video en streaming, archivos de texto, documentos, metadatos y otros) ha tenido un crecimiento explosivo en los últimos años. Esto es debido a que las interacciones y comunicaciones de los usuarios de las redes sociales proveen información clave y fiable que puede ser usada por organizaciones gubernamentales y no gubernamentales en muchos escenarios. Por ejemplo para recolectar información en eventos a gran escala como incendios, terremotos y otros desastres, cada uno de los cuales tienen un impacto ya sea a nivel local, nacional o incluso internacional [1, 2]. A escala individual, los datos generados por el usuario pueden proveer información confiable sobre qué es lo que sucede alrededor de cada usuario, de esta forma ellos pueden usar esta nueva información y tomar decisiones casi en tiempo real.

Las redes sociales (principal fuente de datos de la *social media*) manejan ingentes cantidades de datos, debido a un consistente crecimiento en su número de usuarios. A noviembre de 2014. Twitter tenía 284 millones de usuarios, un incremento del 26% en comparación al mismo periodo el año anterior. Ese mismo mes, Pinterest, Tumblr e Instagram crecieron 57%, 45% y 36% respectivamente [3], indicando una tendencia de preferencia de los usuarios hacia las redes sociales cuyo contenido principal son imagines. Algunas redes sociales, bajo ciertas restricciones, ponen estos datos a disposición de los desarrolladores a través de sus propias APIs[1], para fines de investigación y desarrollo.

La data de las redes sociales se caracteriza por ser vasta, ruidosa, distribuida, desestructurada, y dinámica por naturaleza [4]. Tal vez sea este el motivo por el

[1] API: acrónimo de Application Programming Interfaces

que los métodos tradiciones de análisis y de almacenamiento de datos en servidores centrales, han demostrado ser costosos e ineficientes con estos tipos de datos [5]. Para procesar esta vasta cantidad de datos es necesario el uso de plataformas especializadas en *Big Data*, generalmente se requiere de un *framework* que además de implementar un procesado rápido también permita la integración de algoritmos de Machine Learning.

Algunos de los framewoks de Big Data más extendidos actualmente son Hadoop, HBase y Google Bigtable; que si bien son altamente escalables, están orientados al procesamiento por lotes (*batch* en inglés) lo cual crea ciertas limitaciones en la velocidad con la que los datos son procesados. Sin embargo recientemente han hecho su aparición frameworks con capacidades de procesamiento en *Stream*. Propuestas como Apache Storm, Apache S4 y Apache Spark representan esta nueva oleada. De todos ellos, destaca claramente Apache Spark debido a que su diseño cubre un amplio rango de cargas de trabajo que anteriormente requerían sistemas distribuidos separados, incluidas aplicaciones en batch, algoritmos iterativos, consultas interactivas y streaming [6]. Desarrollada ooriginalmente como un Proyecto de investigación de AMPLab de la UC Berkeley's, Apache Spark es capaza de realizar analíticas en memoria sobre datos en Stream, logrando tiempos de computo 100 veces más rápido que Hadoop MapReduce, o 10 veces más rápidos para los trabajo en disco [7]. Es además más rápido que Storm y S4.

Por su parte, los algoritmos de Machine Learning permiten ver patrones y relaciones entre grandes conjuntos de datos. Existen diversos algoritmos que pueden ser empleados, cuyo output depende del tipo de algoritmo que se utilice. Por ejemplo construir modelos predictivos precisos, o deducir si una nueva pieza de información se relaciona con otra. El uso del machine learning viene desde el inicio mismo de la informática, sin embargo, su uso in ambientes distribuidos para procesar grandes volúmenes de datos es una área de investigación reciente. Debido a la variedad de algoritmos, existen en el mercado diversas herramientas cada una de las cuales trata de adaptarse a un determinado problema (clasificación, predicción, optimización, y regresión); siendo la herramienta más representativa Apache Manhout, o más recientemente la librería MLlib, de Apache Spark.

No son pocas las organizaciones de investigación y startups que sacan ventaja de la disponibilidad y potencialidad de los datos de las redes sociales; Twitter es

claro ejemplo de ello, la red de microblogin ha sido una de las redes sociales sobre las que más investigaciones y emprendimientos se han desarrollado últimamente. Sin embargo, nuevas iniciativas, un tanto controversiales, han encontrado gran atractivo en las redes sociales donde se compartes fotos como Instagram [8], considerando que las imágenes pueden proporcionar información muy útil especialmente en el área de marketing digital

1.2. Motivación

El presente trabajo es una prueba de concepto sobre las funcionalidades de Streaming y Machine Learning de la nueva plataforma de Big Data: Apache Spark. Haciendo uso las APIs de los sub-proyectos Spark Streaming y MLlib se busca implementar un pipeline que permita encontrar los Trending Topic (haciendo uso del algoritmo LDA) sobre datos recopilados de la red social Instagram.

Además de permitir una revisión sobre el funcionamiento de Apache Spark, y su interacción con otras soluciones Big Data (Apache Kafka), se busca establecer las bases para futuros trabajos sobre el procesamiento imágenes en streaming.

Dada la naturaleza del proyecto, no se ahondará en la naturaleza matemática de los algoritmos de Machine Learning empleados; así también, se obviará el procesamiento de imágenes debido a que añade complejidad innecesaria al planteamiento inicial, no obstante, se trabajará con los metadatos de las imágenes a fin de entregar un diseño que guarde coherencia con los objetivos planteados y que posibilite su integración a futuros trabajos.

1.3. Esquema del documento

Esta memoria está compuesta por 6 capítulos, a continuación se brinda una breve descripción del contenido de cada uno de estos:

El capítulo 1 establece el contexto y objetivos del presente trabajo, desarrollando a grandes rasgos la importancia de los datos de la social media y las nuevas tendencias y herramientas que se emplean para analizarlos. Finalizando con una breve descripción del contenido de esta tesis.

Por su parte, el capítulo 2 desarrolla la teoría relacionada a cada uno de los programas y algoritmos que fueron empleados en el presente trabajo.

En el Capítulo 3, podrá encontrar un resumen de los trabajos, investigaciones y tecnologías que de una forma u otra influenciaron en la gestación y desarrollo de esta tesis.

El Capítulo 4, comprende la descripción detallada del diseño, desarrollo e implementación de la pipeline propuesta en el apartado 1.2. Posteriormente, en el Capítulo 5, se describirán las pruebas realizadas sobre la implementación, presentado los resultados de las mediciones y comparativas halladas.

Finalmente el Capítulo 6, se expondrán las conclusiones a las que se llegaron tras el trabajo descritos en los capítulos 4 y 5.

2. Background

2.1. Apache Kafka

Apache Kafka es un sistema de mensajería de publicación-suscripción de código abierto, distribuido, dividido, y replicado. Inicialmente desarrollado en LinkedIn, Kafka proporciona una solución tiempo real de publicación-suscripción que permite consumir grandes volúmenes datos ya sean en tiempo real (Streams) o por lotes (Batch).

Apache Kafka busca unificar y el procesamiento offline y online, proporcionando un mecanismo de carga en paralelo en sistemas Big Data, así como la capacidad para dividir el consumo en tiempo real a través de un cluster de máquinas. La

Ilustración 1 muestra un típico escenario de agregación y análisis de Big Data respaldado por Apache Kafka.

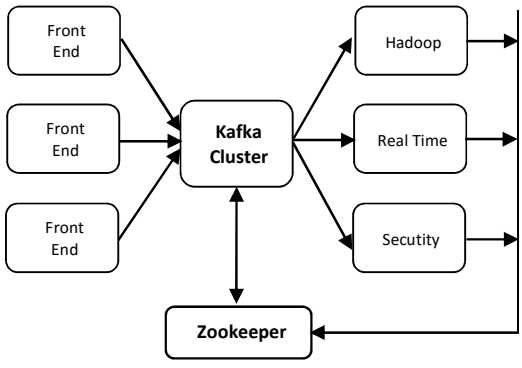

Ilustración 1: Típico clúster de Apache Kafka

Las principales características del Kafka vendrían a ser las siguientes:

- **Mensajería Persistente:** Con Kafka, los mensajes se conservan en el disco mientras se replican dentro del clúster para evitar la pérdida de datos.
- **Alto rendimiento:** Kafka está diseñado para manejar cientos de MB de lecturas y escrituras por segundo, de un gran número de clientes.
- **Distribuido**: Apache Kafka con su diseño centrado en clúster apoya explícitamente la partición de mensajes a través de servidores Kafka y el consumo distribuido sobre un clúster de máquinas *consumer*, manteniendo la semántica de pedidos por partición.
- **Soporte para múltiples clientes**: El sistema Kafka soporta la integración de clientes de diferentes plataformas como Java, .NET, PHP, Ruby y Python.
- **Tiempo real:** Los mensajes producidos por los producer threads deben ser visibles de inmediato a los consumer threads; esta característica es fundamental para los sistemas basados en eventos.

Kafka permite crear tres tipos de clúster:

- A single node – single broker,
- A single node – multiple broker,
- A multiple nodes – multiple broker

Un clúster de Kafka consta de cinco componentes principales:

a) **Topic:** Es una categoría o nombre donde los message producer publican los mensajes. En Kafka, cada tópico consiste en un número configurable de particiones, cada partición está representada por una secuencia inmutable ordenada de mensajes.

b) **Bróker:** Un clúster de Kafka consiste en uno o más servidores donde cada uno de estos puede tener uno o más procesos de servidor ejecutándose, este servidor es conocido como el bróker. Los t*ópicos* son creados en el contexto de los procesos del bróker.

c) **Zookeeper:** ZooKeeper sirve como interfaz de coordinación entre el broker de Kafka y los consumers. Zookeeper fue diseñado para almacenar los datos de coordinación: información de estado, configuración, información de ubicación, y otros.

d) **Producer:** Los Producers publican datos a los *tópicos* escogiendo la partición adecuada dentro del *tópico*. Para equilibrar la carga, la asignación de los mensajes a la partición del *tópico* se puede hacer utilizando round-robin o una determinada función personalizada.

e) **Consumer:** Los *consumers* son las aplicaciones o procesos que se suscriben a los tópicos y procesan los mensajes publicados.

La Ilustración 2 muestra cómo funcionan estos componentes juntos, en este caso se considera un clúster del tipo *single node – multiple broker*, por el que se publica un Topic llamado "test"; note que sólo los *brokers* y los *consumers* utilizan zookeepe para gestionar y compartir el estado

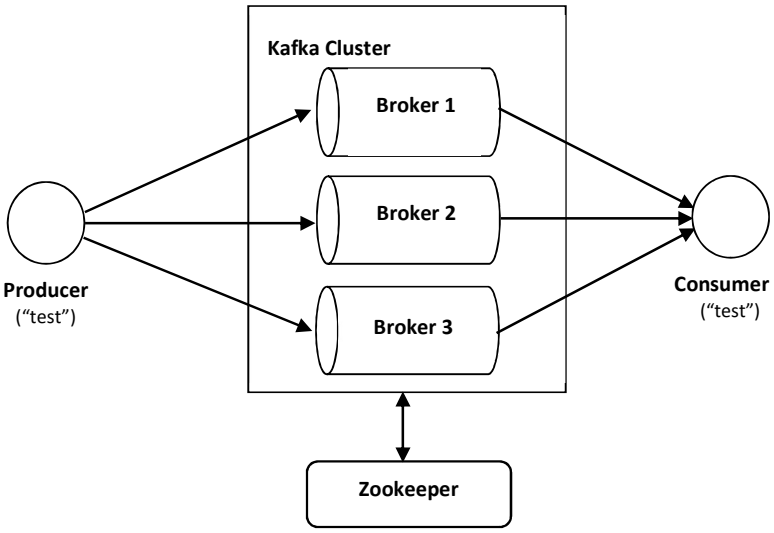

Ilustración 2: Componentes de un clúster de Kafka

Para cada tópico, el clúster de Kafka mantiene un log particionado que luce como la Ilustración 3 muestra:

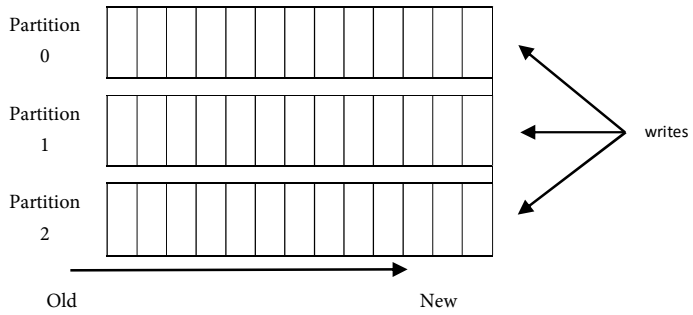

Ilustración 3: Anatomía de un tópico

Cada partición es una secuencia de mensajes ordenada e inmutable que se añade continuamente a un *commit log*. A cada mensaje dentro de las particiones le es asignado un número de identificación secuencial llamado *offset* que lo identifica de manera única.

Cada partición es replicada a través de un número configurable de servidores por la tolerancia a fallos. Cada partición tiene un servidor que actúa como el "líder" y cero o más servidores que actúan como "*followers*". El líder se encarga de todos leer y escribir solicitudes de la partición mientras que los seguidores pasivamente replican al líder. Si el líder falla, uno de los *followers* se convertirá automáticamente en el nuevo líder. Cada servidor actúa como un líder para algunas de sus particiones y como *follower* de otras des esta forma se equilibra la carga dentro del clúster.

2.2. Apache Spark

Apache Spark es una plataforma de computación en clúster de código abierto diseñado para ser rápido y de propósito general. [6] Desarrollado en la UC Berkeley AMP Lab, soporta interfaces de programación para Java, Scala y Python [6]; también se integra con otras herramientas *big data* como clúster de Hadoop y es capaz de acceder a fuentes de datos como HBase, Hive, Cassandra, y HDFS.

A diferencia del paradigma MapReduce de Hadoop. Spark soporta procesamiento en memoria, por lo tanto, los programas con algoritmos iterativos pueden desarrollarse sin tener que escribir un set de resultados después de cada

pasada por los datos, lo que permite un rendimiento hasta 100 veces más rápido que los que escriben en disco [9].

En su *core*, Spark es un "motor computacional" que se encarga de la programación, distribución y monitoreo de aplicaciones que consisten en muchas tareas computacionales a través de muchas máquinas trabajadoras (*worker*), o un clúster de computación.

Como se muestra en la Ilustración 4, en cada clúster de Spark hay un programa conductor (*driver*) que es donde se inicia la ejecución de la lógica de aplicación, mientras que los múltiples *workers* procesan los datos en paralelo. Aunque no es obligatorio, los datos normalmente son colocados con el *worker* y particionados a través del mismo conjunto de máquinas dentro del Clúster. Durante la ejecución, el programa *driver* pasará el código/cierre a la máquina *worker* donde se llevará a cabo el procesamiento de esta partición de datos. Los datos serán sometidos a diferentes pasos de transformación durante su estancia en la misma partición tantos como sean posibles (para evitar el arrastre de datos a través de las máquinas). Al final de la ejecución, las acciones se ejecutarán en el *worker* y el resultado será devuelto al programa *driver* [10].

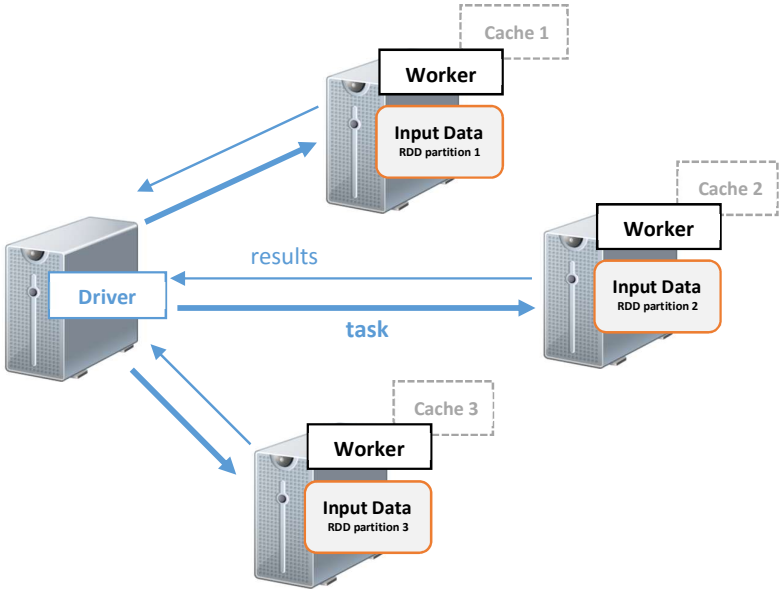

Ilustración 4: Como funciona un clúster de Spark

Para entender la Ilustración 4, usted debe saber lo siguiente:

- Cada nodo *worker* tiene un ejecutor. Un ejecutor es un proceso lanzado por una aplicación, que ejecuta tareas y mantiene los datos en la memoria o disco de almacenamiento a través de ellos. Cada aplicación tiene sus propios ejecutores. Un ejecutor tiene una cierta cantidad de cores o "slots" disponibles para ejecutar las tareas que se le asignen.
- Una tarea es una unidad de trabajo, que será enviada a un ejecutor. Es decir, se ejecuta (parte de) la computación real de una aplicación. Cada tarea ocupa una *slot* o *core* en el ejecutor que lo alberga.

El core engine de Spark faculta múltiples componentes de alto nivel, especializado para diversas cargas de trabajo (Ilustración 5). Estos componentes interoperan estrechamente, permitiendo combinarlos como las librerias en un proyecto de software.

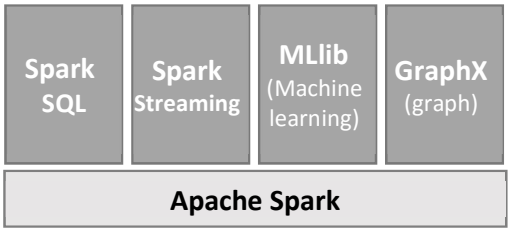

Ilustración 5: Spark Stack

A continuación se presentaran un breve resumen de cada uno de estos componentes.

2.2.1. Componentes

2.2.1.1. Spark Core

Spark Core contiene la funcionalidad básica del Spark, incluyendo componentes para la programación de tareas, gestión de memoria, recuperación de fallos, interacción con los sistemas de almacenamiento entre otros. Spark Core es también el hogar de la API que define Resilient Distributed Datasets (RDDs), que son la abstracción de programación principal de Spark.

Un RDD representa una colección de elementos que pueden ser manipulados en paralelo. Los RDDs sólo se pueden crear a través de operaciones determinísticas

en cualquier dato en un almacenamiento estable (tal como un sistema de archivos compartidos, HDFS, HBase, o cualquier fuente de datos que ofrezca un ingreso en formato Hadoop) u otros RDDs [10].

Un RDD soporta dos tipos de operaciones: *transformaciones*, que crean un nuevo conjunto de datos a partir de uno ya existente, y *acciones,* que son operaciones que devuelven un valor la aplicación o exporta datos a un sistema de almacenamiento.

La Tabla 1 enumera las principales transformaciones y acciones disponibles en Spark RDD

Transformations	map(*f*: T => U) :	RDD[T] => RDD[U]
	filter(*f*: T => Bool) :	RDD[T] => RDD[T]
	flatMap(*f*: T => Seq[U]) :	RDD[T] => RDD[U]
	sample(*fraction* : Float) :	RDD[T] => RDD[T] (Deterministic sampling)
	groupByKey() :	RDD[(K, V)] => RDD[(K, Seq[V])]
	reduceByKey(*f*: (V; V) => V) :	RDD[(K, V)] => RDD[(K, V)]
	union() :	(RDD[T]; RDD[T]) => RDD[T]
	join() :	(RDD[(K, V)]; RDD[(K, W)]) => RDD[(K, (V, W))]
	cogroup() :	(RDD[(K, V)]; RDD[(K, W)]) => RDD[(K, (Seq[V], Seq[W]))]
	crossProduct() :	(RDD[T]; RDD[U]) => RDD[(T, U)]
	mapValues(*f*: V => W) :	RDD[(K, V)] => RDD[(K, W)] (Preserves partitioning)
	sort(c : Comparator[K]) :	RDD[(K, V)] => RDD[(K, V)]
	partitionBy(p : Partitioner[K]) :	RDD[(K, V)] => RDD[(K, V)]
Actions	count() :	RDD[T] => Long
	collect() :	RDD[T] => Seq[T]
	reduce(*f*: (T; T) =>) T) :	RDD[T] => T
	save(path : String) :	Outputs RDD to a storage system, e.g., HDFS

Tabla 1: trasformaciones y acciones para RDD disponibles en Spark

Cada RDD tiene: un conjunto de *particiones*, que son piezas atómicas del conjunto de datos; un conjunto de dependencias en RDDs *parents* que capturan su linaje; una función para computar el RDD en base a sus *parents*; y metadatos sobre su esquema de partición y colocación de datos.

Las dependencias se clasifican en dos tipos: *narrow dependencies*, donde cada partición del RDD *child* depende de un número constante de particiones del *parent* (no proporcional a su tamaño); y las *wide dependencies*, donde cada partición del *child* puede depender de los datos de todas las particiones del *parent* [11]. Esta distinción permite mejoras de procesamiento y una eficiente recuperación después de un fallo de nodo.

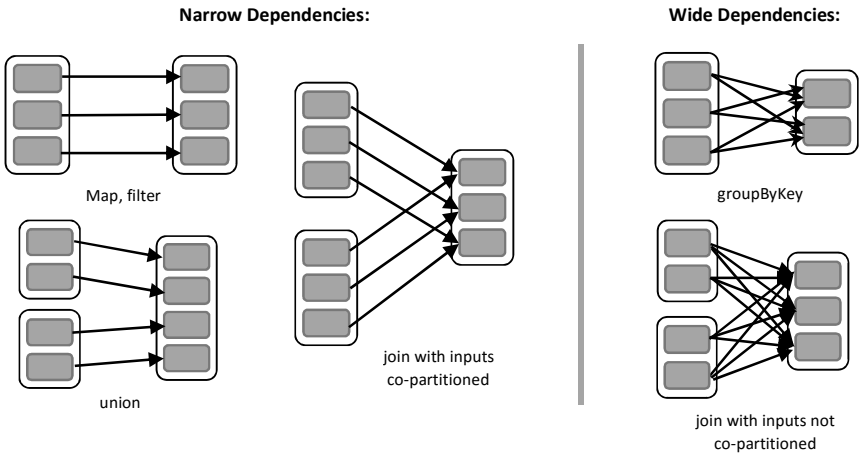

Ilustración 6: Ejemplos de narrow y wide dependencies

Por último, una de las capacidades más importantes de Spark es la persistencia (o caching) de un conjunto de datos en memoria a través de las operaciones. Cuando se persiste un RDD, cada nodo almacena cualquier partición de la misma que se computa en la memoria y los reutiliza en otras acciones en ese conjunto de datos (o conjuntos de datos derivados de éste). Esto permite que a futuras acciones ser mucho más rápidas (a menudo por más de 10 veces). El caching es una herramienta clave para los algoritmos iterativos y uso interactivo rápido.

2.2.1.2. Spark SQL y DataFrame

Spark SQL proporciona soporte para interactuar con Spark a través de SQL, HiveQL (Hive Query Language) o Scala. Representa tablas de bases de datos como un RDD de Spark y interpreta consultas SQL a Spark para ser ejecutadas. Spark SQL implementa un nuevo tipo de RDD llamada Data Frame (conocido también como SchemaRDD en versiones anteriores a la 1.3), que proporciona soporte para datos estructurados y semi-estructurados. Un SchemaRDD se puede crear desde un RDD existente, un Parquetfile, un dataset JSON, o ejecutando HiveQL con los datos almacenados en Apache Hive.

2.2.1.3. Spark Streaming

Spark Streaming es un componente de Spark que permite el procesamiento *live streams* de datos. Spark Streaming proporciona una API para la manipulación *streams* de datos que coinciden estrechamente con la API RDD del *core* de Spark,

por lo que es fácil moverse entre aplicaciones que manipulen datos almacenados en memoria, en el disco, o llegando es tiempo real.

Internamente, Spark Streaming recibe *live streams* como entrada y divide los datos en *batches*, que luego son procesados por el motor de Spark para generar el *stream* final de resultados en *batches*.

Ilustración 7: Interacción entre Spark Streaming y el core

La Ilustración 8 muestra la ejecución de Spark Streaming dentro de los componentes driver-worker de Spark. Para cada fuente de entrada, Spark Streaming lanza receptores (*receivers*), que son las tareas que corren en los ejecutores (*executors*) de la aplicación, recogen los datos de la fuente de entrada y los guardan como RDDs. Estos reciben los datos de entrada y los replican (por defecto) a otro *executor* por lo de la tolerancia a fallos. Estos datos se almacenan en la memoria de los *executors* de la misma forma que los RDDs en caché. Luego el *StreamingContex*t en el programa driver ejecuta periódicamente los trabajos (*jobs*) de Spark para procesar estos datos y combinarlos con RDDs de pasos anteriores.

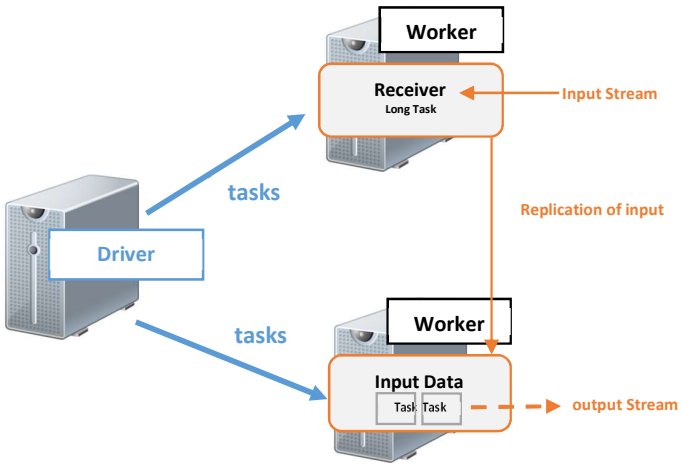

Ilustración 8: Ejecución de Spark Streaming en los componentes de Spark

Spark Streaming proporciona una abstracción de alto nivel llamado *Discretized Stream* DStream. Un Dstream es una secuencia (una serie de RDDs) de datos que llegan a través del tiempo [6]. Los datos de entrada recibidos durante cada intervalo son guardados de forma fiable a través del clúster para formar un conjunto de datos de entrada para ese intervalo. Una vez que el intervalo de tiempo se complete, este conjunto de datos es procesado a través de operaciones paralelas deterministas para producir nuevos conjuntos de datos que representan los resultados del programa o un estado intermedio. [7] [12]

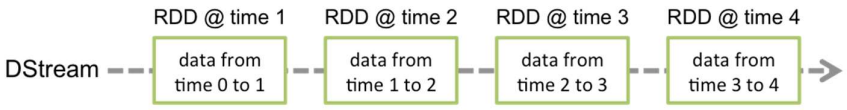

Ilustración 9: Representación de un DStream

Los DStreams pueden crearse desde varias fuentes de entrada, aquellas conocidas como *Basic Sources* que son fuentes directamente disponibles desde la API del StreamingContext, tales como sistemas de archivos, conexiones de socket y actores Akka; y las *Advanced sources*, que son fuentes disponibles que requieren la implementación de enlaces a dependencias adicionales, como Kafka, Flume, Kinesis, Twitter, etc. [11]

Los DStreams proporcionan operadores de *transformación* y *de salida* (*output*) para que los usuarios construyan programas *streaming*, los operadores *output* especifican lo que finalmente se harán con los datos transformados de un s*tream*, mientras que los operadores de *transformación*, producen un nuevo Dstream de unp o más parent streams. A su vez las transformaciones se pueden clasificar en *sin estado* (*stateless*), donde el procesamiento de cada *batch* no depende de los datos de *batches* anteriores) o *con estado* (*stateful*), donde se usan los datos o los resultados intermedios de *batches* anteriores para computar los resultados del batch actual. La Tabla 2 muestra algunos ejemplos de cada operador.

Stateless Transformations	map(f : T => U) : filter(f : T => Bool) : flatMap(f : T => Seq[U]) : union() : join() : reduceByKey(f : (V; V) => V) : repartition() : countByValue() : cogroup() :	DStream[T] => DStream[U] DStream[T] => DStream[T] DStream[T] => DStream[U] (Dstream[T]; Dstream[T]) => Dstream[T] Dstream[(K, V)]; Dstream[(K, W)]) => Dstream[(K, (V, W))] DStream[(K, V)] => DStream[(K, V)] N/A Dstream[(K)] => Dstream[(K, Long)] Dstream[(K, V)]; Dstream[(K, W)]) => Dstream[(K, Seq[V], Seq[W])]
Ouputs	print() : saveAsTextFiles(prefix, [suffix]) : saveAsObjectFiles(prefix, [suffix]): saveAsHadoopFiles(prefix, [suffix]): foreachRDD(f):	Prints first ten elements on screen Save DStream's contents as a text files Save DStream's contents as a SequenceFile of serialized Java objects Save DStream's contents as a Hadoop file Applies a function, f, to each RDD generated from the stream

Tabla 2: operadores transformaciones y ouput de Dtream disponibles en la API de Spark Streaming

Debe considerarse que internamente cada Dstream se compone de múltiples RDDs por lo que cada transformacion stateless se aplica por separado a cada RDD. Además, DStreams introducen nuevas transformaciones stateful que trabajan sobre múltiples intervalos. Éstas incluyen:

- Windowing: el operador window agrupa todos los registros de un rango de intervalos de tiempo en un solo RDD. Como se muestra en la Ilustración 10, cada vez que una ventana se desplaza(*slide*) sobre un Dstream fuente, los RDDs de origen que caen dentro de la ventana son combinados y operados para producir los RDDs del Dstream ventana

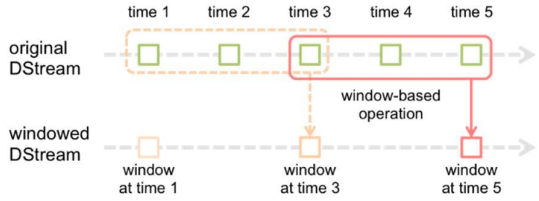

Ilustración 10: Operados window aplicado sobre las últimas 3 unidades de tiempo

- Incremental aggregation: Se usa en el caso de requerir computar un valor agregado a través de una sliding window. El tipo más sencillo sólo usa una

operación asociativa "merge" para combinar los valores. Una versión más eficiente además tiene una función para valores “subtracting” y actualizaciones de estado incrementales.

- Time-skewed joins: los usuarios pueden unir un stream a sus propias RDDs pasadas para computar tendencias.

2.2.1.4. MLlib

MLlib es la libreria de machine learning de Spark. Proporciona múltiples tipos básicos de algoritmos machine learning, incluyendo clasificación binaria, regresión, clustering, collaborative filtering y dimensionality reduction, así como funcionalidades de apoyo como model evaluation y data import. También proporciona algunas primitivas ML de bajo nivel incluyendo un algoritmo genérico de optimización gradient descent.

2.2.1.5. GraphX

GraphX es el motor de grafos de Spark, añadido en la versión 0.9 proporciona una API basada en Pregel para manipular grafos (por ejemplo, un grafo de amistades en una red social) y que realiza cálculos de estos grafos en paralelo. Permite a los usuarios cargas interactivas, transformar y computar grafos masivos. Al igual que Spark Streaming y Spark SQL, GraphX extiende la API RDD de Spark, permitiendo crear un grafo dirigido con propiedades arbitrarias anexas cada vértice y el borde.

2.2.2. Construir Spark

Spark cuenta con paquetes pre-construidos para versiones específicas de Hadoop, MapR y CHD; sin embargo, se recomienda la construcción a partir de los binarios del programa, ya permite una construcción personalizada y/o completa del programa.

El proceso de instalación de Apache Spark resulta muy intuitivo, no obstante, es necesaria la instalación de determinados programas como prerrequisito para que la instalación se dé correctamente. La lista de los programas necesarios para

instalar la versión 1.1, en un ordenador con sistema operativo Ubuntu, es la siguiente:

- Oracle's JDK7
- Scala 2.10
- GIT
- SBT (o MAVEN)

Las versiones de los programas deben ser consideradas, ya que en pueden generar conflictos posteriores debido a la incompatibilidad entre estos y la versión de Apache Spark con la que se esté trabajando, e.g. Apache Spark v 1.1 trabaja con JDK7 y Scala 2.10, a partir de la versión 1.2 es necesario JDK8, y la versión 1.3 admite Scala 2.11. y requiere de Maven 3.0.4 para ser construida;

Por otra parte, se han notado que existen conflictos entre JDK y Open JDK (que viene por defecto en las versiones de Ubuntu superiores a la 12), por lo que se recomienda la desinstalación de este último; así también es necesario hacer explicita la variable de entorno JAVA_HOME.

Si se trabaja con binarios la construcción se realizará con SBT o MAVEN, a través de comandos con opciones adicionales para personalización; en este caso optamos por trabajar con SBT, siendo el siguiente el comando para iniciar una construcción del programa.

```
./sbt assembly
```

2.2.3. Despliegue

Una vez instalados los prerrequisitos y construido Spark se puede poner en funcionamiento el programa. Como se mencionó anteriormente Spark trabaja con un Master y de 1 a más Workers como lo muestra la Ilustración 4. El despliegue de Spark no se encuentra limitado únicamente a redes coordinadas por YARN o MESOS, ya que tambien provides a simple standalone deploy mode. You can launch a standalone cluster either manually, by starting a master and workers by hand, or use our provided launch scripts. Like in this case, is possible to run these daemons on a single machine for testing.

Todo el proceso de despliegue se realiza a través de la terminal de comandos del sistema; La forma más sencilla de iniciar el clúster de Spark, se deberá ubicarse en la carpeta de instalación del programa y llamar al script que inicia

```
./sbin/start-all.sh
```

Esto iniciará el clúster, considere que es necesario configurar antes el nro de *cores* que se utilizarán, pues por cada *core* se desplegará un *worker*, debiendo escogerse un número inferior al de los *cores* disponibles para evitar la saturación de la memoria. La configuración del número de workers podrá realizarse modificando el fichero `config/spark-env.properties` en la carpeta de instalación de Spark

El *master* de Spark requiere de un SHH sin contraseña para acceder a sus *workers*, de esta forma se evita el engorro de estar digitando la contraseña de usuario cada vez que se requiera realizar una acción con Spark.

Una vez finalizada la configuración y desplegado el clúster de Spark, es posible acceder a la interfaz UI de Spark, a través de la URL `http://localhost:8080/`

2.3. Topic Modeling

Es un conjunto de algoritmos estadísticos que tienen como objetivo descubrir y anotar grandes cantidades de documentos con información temática. [13] Un algoritmo *topic model* descubre términos emergentes (tópicos) que se producen en una gran colección no estructurada de documentos mirando donde se posiciona la frecuencia del término en la en la distribución general del número de documentos que contienen dicho término. Una vez que se encuentran los términos emergentes, estos se agrupan utilizando un modelo probabilístico de coocurrencia. [1]

Los documentos son observables mientras la estructura del tópico – los tópicos, la distribución tópicos por documentos, y las asignaciones de tópicos por palabra por documento – se mantiene oculta. El problema computacional central para el topic modeling es el uso de documentos observados para inferir la estructura oculta de los tópicos.

La utilidad de los topic models radica en la propiedad de semejanza entre la estructura oculta inferida y a la estructura temática de la colección. Esta estructura oculta interpretable anota cada documento en la colección - una tarea muy difícil de realizar a mano - y estas anotaciones pueden ser usadas para apoyar tareas como recuperación de información, clasificación, y exploración de corpus. De esta forma, topic modeling provee una solución algorítmica para administrar, organizar y anotar archivos de texto grandes.

Una de las características más importantes de los topic models es que no requieren anotaciones previas o el etiquetado de los documentos, los tópicos surgen del análisis de los textos originales.

La detección de tópicos puede realizarse también de forma no probabilística a través de enfoques que poseen sus propias particularidades y metodologías. A continuación se presenta una breve descripción de estos enfoques:

2.3.1. Document Pivot approaches

Este enfoque típicamente se calcula una medida de similitud entre cualquier par de documentos o un documento y la representación de un clúster prototipo. En el primer caso, si la similitud entre el documento entrante y el documento que mejor coincide que ya está en la colección, se está por encima de cierto umbral, entonces el documento entrante se añade al mismo clúster como el mejor documento coincidente. En caso contrario, se genera un nuevo clúster. Del mismo modo, en el segundo caso, si la similitud del documento entrante a la mejor coincidencia de clúster está por encima de cierto umbral, el elemento se añade al clúster, de lo contrario se crea un nuevo clúster. Típicamente se diferencian en que se calculan la similitud de diferentes maneras, se aplican técnicas especiales para encontrar el mejor elemento coincidente o el clúster, o se utiliza alguna etapa de post-procesamiento.

En general, este tipo de enfoques de clusterización incremental requieren de un ajuste adecuado de un parámetro umbral. Si el umbral es demasiado bajo, entonces un solo clúster representará tópicos demasiado genéricos o una mixtura de tópicos. Si el umbral es demasiado alto, entonces los clústeres tienden a ser fragmentados y un solo tema será representado por varios de ellos. El umbral en la mayoría de los casos es establecido empíricamente y depende de la medida de similitud seleccionado. Hay que señalar que la fragmentación es el problema más

general para enfoques document-pivot. Es decir, es probable que, debido al hecho de que un tópico puede ser expresado de diferentes maneras, muchos de estos grupos representan un solo tópico. Sin embargo, ya que con el tiempo la representación de un tópico se enriquece, es posible aplicar un procedimiento de fusión.

2.3.2. Feature Pivot Methods

Los métodos feature-pivot intentan agrupar términos de acuerdo a sus patrones de coocurrencia. En general, el primer paso de estos enfoques consiste en la selección de los términos que se agruparan (en un clúster) y para los cuales se calculan los patrones de coocurrencia. La selección puede realizarse utilizando diferentes criterios. Por ejemplo, los términos más comunes que no sean palabras vacías o términos emergentes. En el segundo paso, alguna forma de similitud entre términos, típicamente entre pares de términos, es calculada y utilizada en conjunción con algún procedimiento de clusterización. Los enfoques que han aparecido presentan una gran variedad de formas de seleccionar el conjunto de términos que se agrupan, para calcular la similitud entre términos y para llevar a cabo la clusterización.

En resumen, la mayoría de los métodos feature-pivot, sin importar el mecanismo de selección término empleado, examinan los patrones de coocurrencia (de diversas formas) entre pares de términos. En la práctica, dependiendo del algoritmo de clusterización, puede conducir a que términos comunes sean agrupados incorrectamente con algunos términos, simplemente debido al hecho de que tan comunes que pueden ser vinculados a un gran número de tópicos.

2.3.3. Probabilistic topic models

Este enfoque representa la distribución conjunta de tópicos y términos mediante un modelo probabilístico generativo, que consiste en un conjunto de variables latentes que representan tópicos, términos, hiperparámetros, etc. Dos topic models probabilísticos muy conocidos que se han extendido en muchos formas son Probabilistic Latent Semantic Analysis (PLSA) y Latent Dirichlet Allocation (LDA). LDA es probablemente el tópico model más popular; utiliza

variables ocultas que representan la distribución términos por tópico y la distribución tópicos por documento. El aprendizaje y la inferencia de LDA se realizan usando generalmente Variational Bayes, pero han aparecido otros enfoques como el uso de Gibbs Sampling.

3. Trabajo Relacionado

3.1. Soluciones Existentes

Las soluciones de análisis de Big Data surgen en años recientes, como evolución natural de las herramientas de Data Mining, creando y extendiendo el campo de la Data Science.

Las herramientas de Machine Learning, por su parte, datan de mucho antes; en ambos casos, el enfoque de la Computación Distribuida hizo que patente la necesidad de mejorar los algoritmos ya existentes para adaptarlos a esta nueva forma de trabajo dando pie al surgimiento de nuevos algoritmos capaces de aprovechar la eficiencia y potencia de cómputo de los sistemas distribuidos.

Algunas de las herramientas de análisis Big Data en streaming de mayor relevancia actualmente son:

- **Storm:** Storm es un sistema de procesamiento de big data de código abierto desarrollado por Twitter que se caracteriza por su diseño capaz de realizar procesamiento en tiempo real distribuido y es independiente del lenguaje. Storm esta implementado en Clojure y soporta la construcción de topologías que transforman secuencias de datos sin finalizar. Esas transformaciones, a diferencia de los trabajos de Hadoop, nunca se detienen, sino que continúan procesando datos conforme van llegando. Es además es un ejemplo de un sistema complejo de procesamiento de eventos (CEP) capaz de orientarse a la computación o a la detección
- **Spark Streaming:** Es una librería de Spark que aprovecha capacidad de programación rápida core de Spark para realizar analítica streaming. Se ingresan datos en mini- batches y realiza transformaciones RDD en esos mini- batches de datos. Este diseño permite que el mismo código escrito

para el análisis en batch pueda ser utilizado en analítica streaming, en un solo motor.

- **Apache S4:** S4 (Simple Scalable Streaming System) es una plataforma de propósito general, distribuida, escalable, parcialmente tolerante a fallos y conectable, que permite el desarrollo de aplicaciones para el procesamiento continuo e ilimitado de streams de datos. Eventos de datos clave son dirigidos por afinidad a los elementos de procesamiento (PE), que consumen los eventos y realizan una o ambas de las siguientes opciones: (1) emite uno o más eventos que pueden ser consumidos por otros PEs, (2) publica resultados. La arquitectura se asemeja al modelo de Actores, proporcionando semánticas de encapsulación y transparencia de ubicación, lo que permite que las aplicaciones sean masivamente concurrentes.

La Tabla 3 muestra una comparativa entre las capacidades de procesamiento de estas herramientas, apréciese la gran diferencia entre Spark Streaming y las otras soluciones.

Framework	Capacidad de procesamiento
Storm	10,000 records/s/node
Spark Streaming	400,000 records/s/node
Apache S4	7,000 records/s/node
Other Commercial Systems	100,000 records/s/node

Tabla 3: Comparativa entre las principales herramientas streaming

En el caso de Machine Learning, son pocas las herramientas que permiten trabajar en clúster, destacando principalmente:

- **Apache Mahout:** Mahout es un libreria open source de Apache Software Foundation (ASF) cuyo objetivo principal es la producción de implantaciones libres de algoritmos escalables de machine learning usando el paradigma MapReduce. Mahout contiene implementaciones para collaborative filtering, clustering y classification. Mahout también proporciona librerías de Java para operaciones matemáticas comunes (centradas en el álgebra lineal y estadísticas) y colecciones de primitivas Java.

- **Vowpal Wabbit:** (aka VW) es un sistema, librería y programa open source desarrollado originalmente por Yahoo! Research como una aplicación escalable de la machine learning en línea y como apoyo a una serie de reducciones de machine learning, ponderación de importancia, y una selección de funciones de pérdida y optimización de algoritmos.

- **MLlib.**- MLlib es la librería de machine learning de Apache Spark, debido a la arquitectura distribuida de Spark basada memoria es hasta diez veces más rápida que Apache Mahout e incluso mejor que las escalas Vowpal Wabbit. Por ahora, implementa algunos algoritmos estadísticos y de machine learning, entre los que se incluyen: summary statistics, classification and regression, collaborative filtering, clustering, dimensionality reduction, feature extraction and transformation and optimization primitives.

3.2. Artículos

Existen trabajos realizados en los que de una forma u otra se encuentran relacionados con el que se describe en esta memoria, ya sea por que guardan cierta similitud en la temática de su contenido y o por las tecnologías que emplea, aquellos que fueron considerados de mayor relevancia se listan a continuación.

- En [8] se describe el caso de dos startups: Ditto Labs Inc y Piquora, quienes basaron sus modelos de negocio en el uso de imágenes obtenidas a través de redes sociales como Pinterest, Flickr o Instagram u otros medios. En Ditto Labs, analizan las imágenes en busca de logos, si alguien sonríe en la imagen o el contexto de la escena. En el otro caso, Piquora almacena imágenes durante meses y posteriormente muestran las tendencias que reflejan estas imágenes. Esta información es proporcionada a gente de marketing ya sea para enviar anuncios orientados a determinados segmentos o para realizar investigaciones de mercado. También se plantea el asunto de la privacidad en las redes sociales y los vacíos legales existentes en relación a esta.

- En el trabajo de Aiello y otros [1], ofrece una comparativa de seis métodos de topic detection sobre tres datasets de Twitter relacionados a grandes eventos, con distintas cantidades de datos y que fueron recolectados en distintos periodos de tiempo. Los métodos empleados fueron: Latent Dirichlet Allocation (LDA), Document-pivot topic detection (Doc-p), Graph-based

feature-pivot topic detection (GFeat-p), Frequent pattern mining (FPM), Soft frequent pattern mining (SFPM) y BNgram. Como resultados llegaron que la forma en que factores como la naturaleza del fenómeno considerado, el volumen de actividad en el tiempo, el procedimiento de muestreo y el pre-procesamiento de los datos, afectan en gran medida la calidad de la detección de tópicos, y que estos a su vez dependen del tipo de método de detección utilizado. Otra conclusión a la que se llegó fue que las técnicas estándar de procesamiento de lenguaje natural pueden funcionar bien para los streams sociales en temas muy específicos, pero es necesario del uso de nuevas técnicas para manejar streams más heterogéneos. Así también se halló que el método BNgram, basado en n-gramas, coocurrencia y la categorización de tópicos tipo $df - idft$, logra consistentemente el mejor rendimiento a través de todas las condiciones descritas, siendo por lo tanto el más fiable.

- En [14] se desarrolla el trabajo realizado por Qiu y otros, consistente en la implementación de un método de collapsed Gibbs sampling para el modelo Latent Dirichlet Allocation (LDA) en Spark. El enfoque propuesto divide un dataset en $P * P$ particiones, barajando y recombinado estas particiones en P sub-datasets, donde cada P sub-dataset sólo contiene P particiones, posteriormente cada sub-dataset será procesado uno a uno en paralelo. Tras la implementación, se realizaron experimentos sobre tres dataset: estradas de blog (KOS), papers completos (NIPS), y artículos del NY Times. Se pudo apreciar que a pesar de aumentar el número de iteraciones, este método reduce la sobrecarga de comunicación de datos, hace buen uso de la ejecución iterativa de Spark y devuelve un significativo aumento de velocidad en los datasets de gran escala.

- Wang y otros en [15], muestran que el número de tópicos es un factor clave que puede aumentar significativamente la utilidad de un sistema topic modeling. Concretamente, se muestra que un modelo LDA "grande" con al menos 10^5 tópicos inferidos de 10^9 consultas de búsqueda, puede lograr una mejora significativa en un motor de búsqueda industrial y sistemas de publicidad en línea. Para tal efecto desarrollaron un sistema distribuido llamado Peacock para aprender modelos grandes LDA a partir de big data.

- Por su parte, el trabajo presentado en [16] Tamura y otros utilizan imágenes publicadas a través de las redes sociales para extraer eventos y rastrear tópicos en un stream de documentos que contengan imágenes. Para lograr esto por

proponen un método que integra una técnica de clustering con el algoritmo burst-detection de Kleinberg. Para evaluar el método propuesto, utilizaron el texto de los tweets que contenían imágenes. Los resultados experimentales muestran que el método propuesto cumple con los objetivos básicos que fueron planteados, aunque no se muestran métricas de perfomance ni se describe la tecnología empleada para la implementación.

4. Multimedia Big Data Computing for Trend Detection

4.1. Visión General

4.1.1. Analisis & Diseño

Como se definió con anterioridad el presente trabajo es una prueba de concepto, por ese motivo inicialmente se pensó en un diseño que solo incluyera a las librerías de Apache Spark: Spark Streaming y MLlib, como tecnologías a ser usadas. Además, dadas las consideraciones expuestas en el apartado 1.2., se estimó por conveniente centrar el esfuerzo de recopilación de datos en los metadatos y no en las imágenes de Instagram.

Por otra parte, MlLib no contaba con alternativas para Topic Modeling a octubre de 2014, por lo que se decidió realizar algunas pruebas de integración con el algoritmo de clasificación K-means mientras se estudiaba la posibilidad de elaborar un código propio para este fin.

Estos razonamientos permitieron trazar el diseño inicial que muestra la Ilustración 11, el diseño inicial estaba dividido en 4 etapas, cada etapa cubría a grandes rasgos un componente esencial en la arquitectura de una solución big data [17].

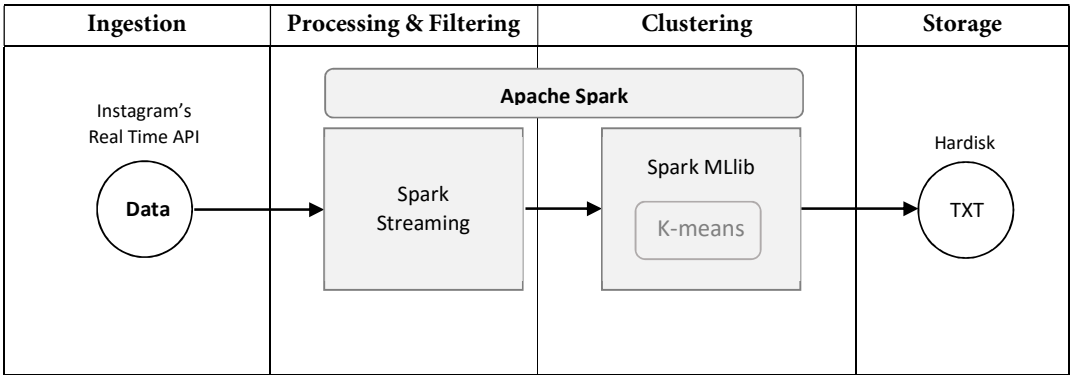

Ilustración 11: Borrador del diseño inicial

A continuación se describen brevemente estas etapas:

a) **Etapa de ingestión de datos.** Comprendería el proceso de captación de la data provista por la API Instagram, tras cumplir con los requisitos exigidos para tal fin.

b) **Etapa de Procesamiento y Filtrado.** Utilizando una conexión personalizada por puerto, los datos serían ingresados a Apache Spark y consumidos como Dstreams. Una vez recibida la data, esta sería tratada de tal forma que se puedan eliminar de la misma, aquellas palabras irrelevantes para el trabajo. El resultado de esta etapa sería una bolsa de palabras.

c) **Etapa de Clustering.** Esta etapa haría uso de la bolsa de palabras generada previamente, la cual sería empleada en la fase entrenamiento y posteriormente en la etapa de clasificación. De acuerdo al razonamiento inicial, los tópicos con mayor frecuencia en cada cluster serían los topic trending resultantes.

d) **Etapa de Almacenaje.** Al final, los datos resultantes serían almacenados en ficheros de texto compatibles con HDFS para para comparativas posteriores.

La investigación conceptual y técnica realizada de forma posterior a esta primera aproximación, permitió esclarecer algunos puntos sobre la idea inicial del trabajo, así como definir nuevas limitantes; a continuación se presentan las conclusiones a las que se llegaron en esta etapa:

- A diferencia de Twitter los datos de las peticiones a Instagram no llegan de forma directa, sino que deben ser recolectados tras recibir las notificaciones de actualización, por lo que resulta necesario descargar los

datos en forma de ficheros JSON; por ende no sería posible ingresar los datos directamente a memoria durante la Etapa de ingestión de datos.

- Al ser ficheros JSON, sería necesario emplear Spark SQL para procesar y filtrar los datos, ya que Spark Streaming únicamente permite manipular datos derivados de ficheros HDFS.
- Al ser uno de los subproyectos más jóvenes (entiéndase que el razonamiento fue realizado usando la versión 1.1. de Apache Spark), MLlib cuenta con un número reducido de algoritmos de Machine Learning, y ninguno diseñado para trabajo en Streaming (la versión 1.3. de Apache Spark introdujo K-means streaming). Esto nos llevó a dos conclusiones, La primera fue la necesidad de buscar un algoritmo - de entre los disponibles - que nos permitiera alcanzar los objetivos planteados; la segunda, era que necesariamente el input de la Etapa de Clustering serían ficheros en disco ante la imposibilidad del trabajo directo en Streaming.

Considerando estos alcances a la par que se iban revisando trabajos previos e investigaciones en curso, resultaba evidente la simplicidad del diseño planteado, además de no cubrir cabalmente con los objetivos planteados.

Se optó entonces por incluir Apache Kafka como un nexo entre las etapas de Ingestión de Datos y la de Procesamiento y Filtrado, esta adición otorgaría la posibilidad de enriquecer la data al permitir ingresar información adicional a la cola Kafka [18], de esta forma se da la posibilidad a futuras investigaciones para ampliar los alcances obtenidos en el presente trabajo; Esta propuesta se desarrolla a mayor profundidad en la Sección 6.2.

También se vio la necesidad de replantear el proceso de captación de datos debido a las particularidades de la API de Instagram para aplicaciones en tiempo real. Al ser obligatorio el trabajo en Batch, se pensó en integrar a la pipeline, programación que permita realizar el proceso de captación automáticamente, pero al trabajar bajo un sistema de notificaciones que se ejecuta periódicamente y con un tráfico variable, acarrearía periodos donde no se reciban datos, motivo por el que descartó esta idea y se optó por realizar el proceso manualmente.

Por otra parte, en noviembre 2014 se inició en los foros de Apache un requerimiento para iniciar el desarrollo del Algoritmo LDA (Latent Dirichlet Allocation) [19] para futuras versiones de Apache Spark, en ese entonces fueron desarrolladas en paralelo hasta tres versiones beta que utilizaban variaciones del Gibbs sampling, sin embargo, fue la versión de Joseph K. Bradley y Xusen Yin basado en [14], la que mayor impulso recibió por parte del equipo de AmpLab. Tras realizar un seguimiento sobre la progresión de esta implementación, y dados los avances del mismo, se optó por incluirlo como parte de este trabajo al ajustarse totalmente a los objetivos del mismo.

Las modificaciones descritas permitieron la mejora del diseño inicial, ajustándolo a las necesidades y objetivos que se buscaban. Así finalmente el diseño de la aplicación quedo como lo muestra la Ilustración 12.

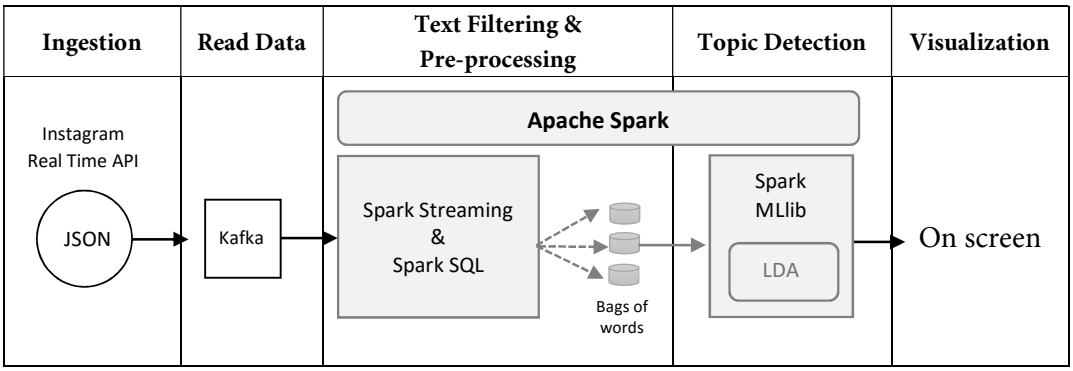

Ilustración 12: Diseño final de la aplicación

Las líneas punteadas representan las ventanas de tiempo en la que son dividas los Dstreams. Como se vio en el apartado 2.2. Spark Streaming trabaja con ventanas de tiempo; estas ventanas son necesarias para continuar con las tareas de procesado de un grupo de datos, ya que si no se limitase el periodo de ingreso de la data el sistema se dedicaría a leer permanentemente y nunca iniciaría el procesado de datos. De acuerdo al diseño en cada ventana se producirá una bolsa de palabras, cada una de las cuales podrán ser utilizadas en la etapa de Topic Detection

4.1.2. Software utilizado

Para el desarrollo del presente trabajo se emplearon esencialmente dos programas: Apache Kafka y Apache Spark, sin embargo el uso e instalación de

estos programas está relacionado a otras tecnologías, algunas de ellas fueron descritas en los apartados 2.1 y 2.2., respectivamente, y otros serán precisados más adelante en la etapa que les corresponda. En este apartado se buscar describir algunas particularidades de Spark y Kafka que son necesarias conocer antes de iniciar con descripción del trabajo realizado a fin de lograr una mejor comprensión de éste.

Durante el desarrollo del este trabajo Apache Spark representó un reto debido a la rápida progresión del programa y la forma de programar una aplicación, algo más compleja de lo que se percibía en un principio. Desde el inicio de este trabajo Apache Spark ha experimentado una rápida progresión en su versionado, iniciando con la version 1.1 en julio 2014; en noviembre 2014 fue lanzada la version 1.2, y más recientemente, en marzo 2015, la version 1.3 fue liberada. En cada lanzamiento se incorporaron mejoras en las librerías que fueron consideradas, adecuando los avances realizados a la progresión de este trabajo.

Las funcionalidades de Apache Spark pueden ser desplegadas a través de la Shell del programa llamada spark-shell, donde básicamente se trabaja ingresando sentencias y/o trozos de código en Scala, que resulta útil para ejemplificar y testear las librerías (excepto Spark Streaming), Todo el material multimedia proporcionado por AMPLab y Databricks (principales promotoras de Spark), hace uso de la Shell sin embargo la programación de una aplicación es algo más compleja.

La programación en Scala puede realizarse en un editor de texto tipo GEDIT sin problema, pero debe seguirse una determinada estructura de carpetas si el código es complejo; programar en Scala para Spark puede realizarse de la misma forma, no obstante la depuración del código puede resultar realmente tediosa, por lo que para el presente trabajo se utilizó Intellij IDEA v 13.004, un framework especializado en JAVA, pero que cuenta con plug-ins que permiten el trabajo fluido con Scala y SBT.

Un programa típico en Scala para Spark tiene la siguiente estructura:

```
|-- build.sbt
|-- assembly.sbt
|-- src/
|   |-- main/
|   |   |-- scala
|   |   |-- resources
|   |-- test/
```

```
|   |   |-- scala
|   |   |-- resources
|   target/
|   |-- scala2.10/
```

- En el fichero build.sbt se listan las *dependencias* que serán proveidas, los repositorios desde donde se obtendrán las extensiones, y alguna otra programación adicional relacionada al funcionamiento de la aplicación.
- Por su parte, en el fichero assembly.sbt se realiza programación necesaria para la construcción de la aplicación.
- Los ficheros con la programación principal del programa deben incluirse en carpeta src/main/scala/ y en la carpeta src/test/scala/ aquellos que contengan código de prueba.

Una vez concluida la programación ésta deberá ser construida. Al ensamblarse el código se creará un fichero jar, también denominado UberJAR debido a la gran cantidad de megas que ocupa en disco, que contiene todas las librerías necesarias para la ejecución de la aplicación; salvo se haya especificado algo distinto, el fichero compilado podrá encontrarse en la carpeta target/scala-2.10/.

Antes ejecutar el programa, previamente deberá ser desplegado el cluster de Spark. Finalmente, se deberá utilizar spark-submit para lanzar el programa, para lograr esto deberemos digitar el commando bin/spark-submit --master --class --jars

Como se puede apreciar spark.submit hace uso de varios parámetros, siendo los más habituales:

```
-- master
-- cluster
-- jar
```

En el caso de Apache Kafka, tras profundizar en el aprendizaje del lenguaje Scala, caímos en cuenta de que tanto Zookeeper como el Kafka Server podían desplegados como parte de la aplicación, bastaba con declarar las dependencias necesarias e incluirse las configuraciones de despliegue para ambos como parte del código de la aplicación, siendo finalmente ésta la forma en la fueron utilizados los programas.

Realizadas estas precisiones, a continuación se ofrece una descripción detallada del trabajo realizado etapas por etapa, siguiendo el orden establecido en el diseño de la aplicación.

4.2. Stage 1: Data Ingest from Instagram

Instagram es una red social cuyo propósito principal es compartir fotos y videos, es accesible principalmente a través de la app del mismo nombre (disponible para sistemas Android e IOs) o de su sitio web desde su sitio web (www.instagram.com), aunque en este último caso las funcionalidades son más limitadas. La aplicación permite editar las fotografías y compartirlas en otras redes sociales como Facebook, Tumblr, Flickr y Twitter. Creada por Kevin Systrom y Mike Krieger, fue lanzada en octubre de 2010, a diciembre de 2014, ha superado los 300 millones de usuarios.

Una publicación regular en Instagram consta de los siguientes elementos: Usuario, ubicación, imagen o video (desde ahora llamado "elemento multimedia"), texto de la publicación, *tags*, *likes* y comentarios, tal y como se muestra en la Ilustración 13. No obstante, es necesario precisar que salvo el usuario y el *elemento multimedia* el resto de elementos es prescindible en una publicación.

En el apartado 1.1 establecimos que Instagram brinda a sus usuarios un acceso abierto a sus a sus datos a través de su propia API, haciendo posible el desarrollo de aplicaciones secundarias y de naturaleza necesariamente distinta a la app [20]. La API de Instagram es capaz de proveer a las aplicaciones notificaciones instantáneas de nuevas fotos en cuanto estas son posteadas, vale decir, en tiempo real. Para conseguir esto, es necesario cumplir previamente con ciertos requerimientos los cuales se describirán a continuación.

Ilustración 13: foto posteada en Instagram

4.2.1. Registro

El primer paso es obtener una cuenta de usuario de Instagram, la cual se puede obtener a través de su app realizando el proceso registro de usuario o vinculando una cuenta de Facebook existente. Una vez obtenida la cuenta, debemos identificarnos en el sitio de desarrollo de Instagram[2] y registrar nuestra aplicación. La API de Instagram emplea el protocolo OAuth2[3] para autenticar e Identificar a sus usuarios, debido a lo cual es necesario el registro de la aplicación; a cada aplicación registrada le será asignado un OAuth client_id y un client_secret únicos, la Ilustración 14 muestra un ejemplo de los datos asignados.

[2] https://instagram.com/developer/

[3] OAuth2 es un protocolo de autorización que permite a terceros (clientes) acceder a contenidos propiedad de un usuario (alojados en aplicaciones de confianza, servidor de recursos) sin que éstos tengan que manejar ni conocer las credenciales del usuario.

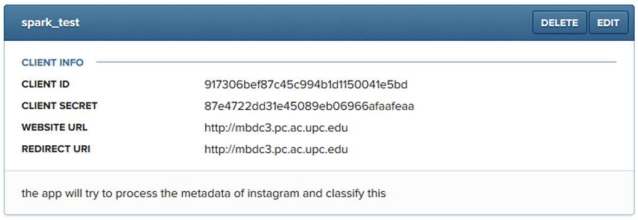

Ilustración 14: datos de cliente asignados

Durante el proceso de registro de la Aplicación serán requeridos 4 campos: Application Name, Description (de la aplicación), Website and a OAuth redirect_uri. La redirect_uri especifica donde serán redirigidos los usuarios después de haber elegido o no autenticarse en una aplicación.

4.2.2. Autenticación

La API de Instagram requiere autenticar y autorizar las aplicaciones en tiempo real por motivos de seguridad y control debido al tipo de peticiones que realizan. Authenticated requests require an access_token. These tokens are unique to a user and should be stored securely, also, are temporal, so is highly probable that we will need repeat this process every so often.

Para recibir el access_token es imperativo hacer lo siguiente

Debe enviarse algunos datos del usuario a la URL de autorización de Instagram. Es necesario enviar el client_id obtenido en el paso previo, y el redirect_uri para pedir el código inicial que permita continuar el proceso, Deberá emplearse un navegador web y escribir como una dirección URL el siguiente comando:

```
https://api.instagram.com/oauth/authorize/?client_id=917306bef87c45c994b1d11
150041e5db&redirect_uri=http://mbdc3.pc.ac.upc.edu&response_type=code
```

Entonces Instagram se redirigirá a la redirect_uri, proporcionada añadiendo el código a la URL

```
http://mbdc3.pc.ac.upc.edu?code=ff3458609296431c815d900f29149839
```

Una vez obtenido este código podremos intercambiarlo por el access_token, para esto será necesario enviar algunos parámetros requeridos mediante el método POST de HTTP, empleando cURL[4], el pedido sería como se muestra a continuación:

```
cURL -F 'client_id=917306bef87c45c994b1d11150041e5db ' \
-F 'client_secret=87e4722dd31e45089eb06966afaafeaa' \
-F 'grant_type=authorization_code' \
-F 'redirect_uri=http://mbdc3.pc.ac.upc.edu' \
-F 'code= ff3458609296431c815d900f29149839' \
https://api.instagram.com/oauth/access_token
```

Si tiene éxito, se retornará el OAuth Token que deberemos usar para hacer requerimientos autenticados a la API, también es enviada la información del usuario que realiza el requerimiento:

```
{
"access_token":"1130302126.76a0be6.ff2f770f1a4749b4a7e1f89ddbe1cd32",
"user": {
      "username":"omar_sulca",
      "bio":"",
      "website":"",
      "profile_picture":"https:\/\/instagramimages-
      a.akamaihd.net\/profiles\/profile_1130702126_75sq_1395405222.jpg",
      "full_name":"Omar Sulca","id":"1142562126"
      }
}
```

Una vez obtenido el acces_token tendremos la facultad de realizar los pedidos de datos que necesitemos, pero antes es necesario realizar una última actividad.

4.2.3. Requests con la API Real Time

Como se mencionó anteriormente la API para aplicaciones en tiempo real de Instagram trabajar un sistema de comunicaciones los servidores de Instagram enviarán un POST al callback URL definido por el usuario cada vez que exista nueva data disponible.

[4] curl is a command line tool and library for transferring data with URL syntax, supporting DICT, FILE, FTP, FTPS, Gopher, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMB, SMTP, SMTPS, Telnet and TFTP

Para recibir las notificaciones es necesario estar suscrito a uno de los cuatro tipos de objetos básicos en Instagram:

- **Users:** recibe notificaciones cuando los usuarios que están registrados en una aplicación de nuevas fotos.
- **Tags:** recibe notificaciones cuando se sube una foto etiquetada con tags elegidos por el usuario.
- **Locations:** recibe notificaciones cuando se suben fotos etiquetadas con una localización específica
- **Geographies:** recibe notificaciones cuando se suben fotos en una localización geográficamente arbitraria definida por un punto central y un radio.

También deberá proporcionarse un callback URL que soporte los métodos GET y POST, esta dirección deberá ser válida (existente) ya que es aquí donde serán enviadas las actualizaciones.

Conocidos estos requisitos y en concordancia con lo expresado en el apartado 4.1, era necesario definir los criterios sobre los cuales se realizarían los request; después de muchas consideraciones se seleccionaron dos casos efectuar el proceso de análisis de datos:

- En el primero caso se buscaría detectar los Topic Trending de un evento a través del tag identificativo, siendo el evento seleccionado la copa mundial de la FIFA 2014, y el tag #worldcup2014.
- En el segundo caso, se buscaría sabes cuáles son los tópicos más tratados por los seguidores de una determinada campaña de marketing y del producto que la realiza. Siendo el mercado de la moda uno de los que mayor expectativas atrae se decidió trabajar con data relacionada a Desigual©, empleando para ello los tags #lavidaeschula y #desigual.

Entonces, una vez definido que el trabajo se realizaría usando suscripciones al objeto Tags, se definió que la callback URL seria http://mbdc3.pc.ac.upc.edu, dirección perteneciente a un servidor virtual proporcionado por la facultad.

Con los requerimientos cubiertos, ahora debíamos crear las suscripciones correspondientes, para tal fin se debía enviar un POST como el que se muestra a continuación:

```
cURL -F ' client_id=917306bef87c45c994b1d11150041e5db ' \
     -F 'client_secret=87e4722dd31e45089eb06966afaafeaa' \
     -F 'object=tag' \
     -F 'aspect=media' \
     -F 'object_id=lavidaeschula \
     -F 'callback_URL=http://mbdc3.pc.ac.upc.edu/' \
     https://api.instagram.com/v1/subscriptions/
```

En respuesta, la API de Instagram envía a la callaback URL un GET, que no es visible para el desarrollador, de acuerdo a la documentación el GET luce de la siguiente manera:

```
http://mbdc3.pc.ac.upc.edu/?hub.mode=subscribe&hub.challenge=15f7d1a91c1f40f8
a748fd134752feb3&hub.verify_token=myVerifyToken
```

Para continuar es necesario recibir la variable hub.challenge y devolver el valor que contenga (esto se puede lograr usando un sencillo script en php que imprima dicha variable). Aquí surge un problema debido a un error en la documentación ya que la variable se denomina hub_challenge, al corregir esto finalmente se recibe la notificación de la suscripción como la siguiente:

```
{
   "meta": {
   "code":200
   },
   "data":  {
            "object":"tag",
            "object_id":"lavidaeschula",
            "aspect":"media",
            "callback_URL":"http:\/\/mbdc3.pc.ac.upc.edu\/",
            "type":"subscription",
            "id":"37623555"
            }
}
```

El mismo procedimiento fue realizado en el caso de los tags #desigual y #worldcup2014. Finalizado el procedimiento solo resta configurar el servidor para

recibir los POST con las actualizaciones. El cuerpo de las notificaciones contiene texto en formato JSON, muy similar al siguiente fragmento:

```
{
    {
        "subscription_id": "1",
        "object": "tag",
        "object_id": "worldcup2014",
        "changed_aspect": "media",
        "time": 1297286541
    },
    {
        "subscription_id": "2",
        "object": "tag",
        "object_id": "lavidaeschula",
        "changed_aspect": "media",
        "time": 1297286541
    },
    ...
}
```

Recibida la notificación podremos realizar los request para recibir y almacenar los datos, para esto se hará uso de los endpoints[5] de la API de Instagram. Los endpoints son accesibles vía **https** y están localizados en **api.instagram.com**; muchos request se pueden realizar únicamente con el client ID, este no es el caso si se busca recibir datos de suscripciones a objetos Tag, siendo necesario para ello poseer un access_token como el que se obtuvo en el apartado 4.2.2. El request vendría a ser algo como lo siguiente:

```
https://api.instagram.com/v1/tags/lavidaeschula/media/recent?access_token=113030212
6.76a0be6.ff2f770f1a4749b4a7e1f89ddbe1cd32
```

Cabe recordar que existe un límite máximo de 5000 request por hora por access_token.

En respuesta la API enviará un fichero de texto en formato JSON con los datos solicitados; el esquema general de los ficheros JSON obtenidos se muestra a continuación:

[5] Endpoint, the entry point to a service, a process, or a queue or tópico destination in service-oriented architecture

```
Root
 |-- attribution: string (nullable = true)
 |-- caption: struct (nullable = true)
 |    |-- created_time: string (nullable = true)
 |    |-- from: struct (nullable = true)
 |    |    |-- full_name: string (nullable = true)
 |    |    |-- id: string (nullable = true)
 |    |    |-- profile_picture: string (nullable = true)
 |    |    |-- username: string (nullable = true)
 |    |-- id: string (nullable = true)
 |    |-- text: string (nullable = true)
 |-- comments: struct (nullable = true)
 |    |-- count: long (nullable = true)
 |    |-- data: array (nullable = true)
 |    |    |-- element: struct (containsNull = true)
 |    |    |    |-- created_time: string (nullable = true)
 |    |    |    |-- from: struct (nullable = true)
 |    |    |    |    |-- full_name: string (nullable = true)
 |    |    |    |    |-- id: string (nullable = true)
 |    |    |    |    |-- profile_picture: string (nullable = true)
 |    |    |    |    |-- username: string (nullable = true)
 |    |    |    |-- id: string (nullable = true)
 |    |    |    |-- text: string (nullable = true)
 |-- created_time: string (nullable = true)
 |-- filter: string (nullable = true)
 |-- id: string (nullable = true)
 |-- images: struct (nullable = true)
 |    |-- low_resolution: struct (nullable = true)
 |    |    |-- height: long (nullable = true)
 |    |    |-- URL: string (nullable = true)
 |    |    |-- width: long (nullable = true)
 |    |-- standard_resolution: struct (nullable = true)
 |    |    |-- height: long (nullable = true)
 |    |    |-- URL: string (nullable = true)
 |    |    |-- width: long (nullable = true)
 |    |-- thumbnail: struct (nullable = true)
 |    |    |-- height: long (nullable = true)
 |    |    |-- URL: string (nullable = true)
 |    |    |-- width: long (nullable = true)
 |-- likes: struct (nullable = true)
 |    |-- count: long (nullable = true)
 |    |-- data: array (nullable = true)
 |    |    |-- element: struct (containsNull = true)
 |    |    |    |-- full_name: string (nullable = true)
 |    |    |    |-- id: string (nullable = true)
 |    |    |    |-- profile_picture: string (nullable = true)
 |    |    |    |-- username: string (nullable = true)
 |-- link: string (nullable = true)
```

```
 |-- location: string (nullable = true)
 |-- tags: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- type: string (nullable = true)
 |-- user: struct (nullable = true)
 |    |-- bio: string (nullable = true)
 |    |-- full_name: string (nullable = true)
 |    |-- id: string (nullable = true)
 |    |-- profile_picture: string (nullable = true)
 |    |-- username: string (nullable = true)
 |    |-- website: string (nullable = true)
 |-- users_in_photo: array (nullable = true)
 |    |-- element: string (containsNull = true)
....
```

Si el fichero enviado no basta para mostrar toda la información disponible, la API incluirá el campo Pagination, el cual incluirá el parámetro **next_URL** con la dirección del siguiente set de datos.

4.2.4. Automatización

Tras todo el proceso realizado, era necesario simplificar y automatizar la recolección de datos, con este fin se implementó el algoritmo en PHP que muestra el Algoritmo 1.

```
<?php
if (isset ($_GET['hub_challenge'])){
    echo $_GET['hub_challenge'];
}
else{
    $myString = file_get_contents('php://input');
    $sub_update = JSON_decode($myString);
    $access_token = '{ previously saved, get this from DB or flatfile }';

    foreach($sub_update as $k => $v) // can be multiple updates per call
    {
        $recent_data =
file_get_contents('https://api.instagram.com/v1/tags/'.$sub_update-
>object_id.'/media/recent?access_token='.$access_token&count=100);
        file_put_contents('activity.log', serialize($recent_data->data),
FILE_APPEND | LOCK_EX);
    }
}
?>
```

Algoritmo 1: Automatización de recopilación de data

La primera parte del algoritmo permite responder a la *Verfication Challenge* que envía la API de Instagram, muy necesaria para la suscripción a cualquier tipo objeto. La segunda parte verifica constantemente si se reciben POST en el servidor, si los hubiera, procede a decodificar la información interpretándola en formato JSON; debido a que cada notificación puede incluir actualizaciones en uno o más Tags, revisa el contenido del todo el POST buscando el object_id (en este caso el nombre del Tag) y envía un https request cada vez que encuentra uno, haciendo uso del access_token que lee previamente desde un fichero TXT externo; limitándose cada request a las 100 subidas más recientes. Los datos recibidos son almacenados en el fichero "activity.log" que acumulará toda la data según se vaya actualizando y posteriormente será transformado a un fichero txt.

El algoritmo fue implementado en el fichero index.php, ubicado en la carpeta de publicación del servidor web (/var/www/), por su parte el fichero activity.log se ubico en la carpeta /home/JSONlog/ con permisos de lectura escritura.

La información sobre los data sets recopilados se tratada en el apartado 5.1.2.

4.3. Stage 2: Reading data, from Kafka to Spark

Una vez recolectados los datos, resultaba necesario ingresarlos a la pipeline de Spark utilizando Kafka, siendo necesario para ello realizar un conjunto de actividades secuenciales, iniciando por desplegar un cluster de Kafka, para después configurar un *producer*, enviar los datos a la cola y recibirlos dentro de Spark por medio de un consumer.

Durante el desarrollo de las pruebas fue implementado un cluster de Kafka del tipo single node – single broker, ya que se trabajaba con una única fuente de datos; los datos se publicaron a través del tópico "testing-input", siendo codificados con Apache Avro, y utilizando la Kryo serialization.

Es posible desplegar un cluster de Kafka directamente desde la aplicación, para eso es necesario incluir algunas librerías como org.apache.curator.test.TestingServer para zookeeper, kafka.server.{KafkaConfig, KafkaServerStartable}, para el Kafka bróker y org.I0Itec.zkclient.ZkClient para crear la interacción entre el broker y Zookeeper; además es necesario incluir en la programación la configuración las propiedades básicas del broker. Lo mismo debe

realizarse con los parámetros del *brokerList* del *producer* y crear el t*ópico*. Solo entonces es posible la publicación de los datos.

Al enviar los datos el producer inicia una suboperacion para serializarlos. La serialización es importante ya que influye en el performance de la aplicación, pues debido a su tamaño o formato algunos datos pueden relentizar la computación; Spark proporciona dos librerías para prevenir esto: *Java serialization* (por defecto) y *Kryo serialization*. La librería Kryo es 10 veces más rápida de la librería Java pero no soporta todos los tipos serializables y requiere el *registro* de las clases que han de utilizarse, pese a esto su uso es altamente recomendado [7]. Una de las alternativas existentes para la serialización de datos persistentes es el framework Apache Avro. Avro usa JSON para definir tipos de datos y protocolos, y serializa los datos comprimiéndolos en formato binario, además es registrable por Kryo.

A continuación, se muestra el esquema de la clase JSON que fue empleada en el trabajo:

```
{
  "type": "record",
  "name": "JSON",
  "namespace": "avro",
  "fields": [ {
    "name": "text",
    "type": "string",
    "doc": " The content of the user's
message "
  } ],
  "doc": "A basic schema for storing
Instagram metadata"
}
```

Como se puede apreciar únicamente fue incluido el campo **texto** (mensaje que acompaña a la imagen cuando se publica), pues contiene la información que se busca extraer. La codificación realizada con Avro genera una clase JAVA, la cual podrá referenciada en la función de registro de la Kryo serialization tal y como se muestra a continuación:

```
class KafkaSparkStreamingRegistrator extends KryoRegistrator {
    override def registerClasses(kryo: Kryo) {
    kryo.register(classOf[JSON], AvroSerializer.SpecificRecordSerializer[JSON])
    ()
    }
}
```

Configurada la serialización, el producer leerá los datos a publicar: Nuevamente, al trabajar con ficheros JSON resultó necesario adecuar el código para la lectura de los mismos, dada su estructura y la necesidad de extraer datos específicos (campo texto), se decidió recurrir a la API de Spark SQL para leer los datos, tal como se muestra el siguiente código:

```
…
    val users = sqlContext.jsonFile("/home/user/Documents/Desigual/*.txt")
    users.registerTempTable("db")
    val ins_data = sqlContext.sql("SELECT caption.text FROM db WHERE caption.text
    is not null”)
    ins_data.map(p => p(0).toString)collect.foreach{
           val temp = p
…
     private val inputTopic = KafkaTopic("testing-input")

     val producerApp = {
       val config = {
         val c = new Properties
         c.put("producer.type", "sync")
         c.put("client.id", "kafka-spark-streaming-test-sync-producer")
         c.put("request.required.acks", "1")
       }
       kafkaZkCluster.createProducer(inputTopic.name, config).get
     }
…
     messages.foreach {
           case JSON =>
           val bytes = Injection(JSON)
             producerApp.send(bytes)
     }
…
```

Algoritmo 2: Configuración y Puesta en marcha del Producer

Como se aprecia en el Algoritmo 2, Spark SQL permite leer múltiples ficheros JSON, cada fichero es trasformado en un Data Frame (RDD), posteriormente sus valores son registrados en una tabla temporal llamada “db”; al ser tratado como una base de datos relacional, para extraer los datos será necesario realizar una consulta SQL, los resultados de la consulta pasaran a formar parte de la lista message. A continuación, se crean el tópico y el Producer, y se inicia él envió de mensajes

El `KafkaInputDStream` de Spark Streaming, es el “conector” entre Kafka y Spark, y sirve para implementar el consumer. Spark corre un receiver (= task) por input DStream, lo que significa que múltiples DStreams paralelizaran las operaciones de lectura a través de múltiples cores.

Con el Producer enviando información, debíamos definir un Consumer para leer los datos, decodificarlos e iniciar el stream de datos en Spark. Tal y como muestra el Algoritmo 3

```
…
def consume(m: MessageAndMetadata[Array[Byte], Array[Byte]], c:
ConsumerTaskContext) {
 val json = Injection.invert(m.message())
        for {j <- json} {
          logger.info(s"Consumer thread ${c.threadId}: received Data $j from
${m.topic}:${m.partition}:${m.offset}")
        }
      }
      kafkaZkCluster.createAndStartConsumer(inputTopic.name, consume)
…
val streams = (1 to inputTopic.partitions) map { _ =>
KafkaUtils.createStream[Array[Byte], Array[Byte], DefaultDecoder, DefaultDecoder](
            ssc,
            kafkaParams,
            Map(inputTopic.name -> 1),
            storageLevel = StorageLevel.MEMORY_ONLY_SER
          ).map(_._2)
        }
        val unifiedStream = ssc.union(streams)
        val sparkProcessingParallelism = 1
        unifiedStream.repartition(sparkProcessingParallelism)
…
```

Algoritmo 3: Ingreso de datos a Spark

El código demuestra como leer desde todas las particiones del tópico. Se creó una Dstream de ingreso por cada partición del tópico, para después unir esos stream en uno solo.

Es necesario resaltar que en este punto, se configuró el `StreamingContext` para que los datos ingresen en ventanas de tiempo de 5 segundos, lo que significaba que cada 5 segundos se creaban Dstreams, el ingreso de datos se detenía por otros 5, mientras se procesaban los datos y el ciclo volvía a iniciarse.

```
…
val batchInterval = Seconds(5)
…
```

4.4. Stage 3: Text Filtering and Pre-Processing

Para poder pasar a la siguiente etapa, los documentos debían pre-procesarse, desestimando los datos irrelevantes, ya que tanto el texto como los tags en Instagram contienen caracteres innecesarios y puntuación que debe ser removida. En esta etapa, los símbolos y palabras vacías debían ser filtrados pues no proveían ningún beneficio.

```
val tokenized: RDD[Seq[String]] =
texto.map(_.toLowerCase.split("\\s")).map(_.filter(_.length >
3).filter(_.forall(java.lang.Character.isLetter)))

val termCounts: Array[(String, Long)] =
  tokenized.flatMap(_.map(_ -> 1L)).reduceByKey(_ + _).collect().sortBy(-_._2)

val numStopwords = 20
val vocabArray: Array[String] =
  termCounts.takeRight(termCounts.size - numStopwords).map(_._1)
```

Algoritmo 4: Proceso de filtering empleado

En el Algoritmo 4 se aprecia el proceso de filtering realizado; como es sabido, los documentos ingresaron a Spark como datos del tipo String, para crear la bolsa de palabras necesaria en la siguiente fase se precedió a dividir el texto en palabras, remover los términos no alfabéticos, los términos cortos con menos de 4 caracteres y los 20 términos más comunes (como stop words).

Recordemos que el LDA de Spark, no trabaja en streaming, debiéndose almacenar los resultados en ficheros para que fuese posible acceder a ellos posteriormente. En este caso cada Dstream, debía reducirse a RDDs comunes, cada RDD en esta instancia contiene el grupo de datos derivados del pre-procesado realizado, posteriormente fueron almacenados como ficheros txt.

4.5. Stage 4: LDA Trend Detection

Contando ya con los datos necesarios, solo quedaba realizar la implementación del algoritmo LDA para obtener los trending topics.

Como se mencionó en el apartado 2.3 LDA es un modelo estadístico que intenta hallar los tópicos en una colección de documentos. En líneas generales el modelo LDA funciona así: se define formalmente un tópico para hacer una

distribución sobre un vocabulario fijo. Por ejemplo, el tópico *genética* tiene palabras acerca de genética con alta probabilidad y el tópico *biología evolucionaria* tiene palabras acerca de la biología evolucionaria con alta probabilidad. Se asume que esos tópicos son especificados antes que ningún dato haya sido generado, es decir que el algoritmo no tiene información acerca de los tópicos. Después por cada documento en la colección se generarán las palabras en un proceso de dos fases:

1. Se elige aleatoriamente una distribución sobre los tópicos.
2. Por cada palabra en el documento:
    - se elige aleatoriamente un tópico de la distribución sobre tópicos del paso #1
    - se elige aleatoriamente una palabra de la distribución correspondiente sobre el vocabulario

Éste modelo estadístico refleja la intuición de que los documentos contienen múltiples tópicos. Cada documento contiene los tópicos en diferente proporción (paso #1); cada palabra en cada documento se extrae de uno de los tópicos (paso #2b), donde el tópico seleccionado es escogido de la distribución por documento sobre los tópicos (paso #2a).

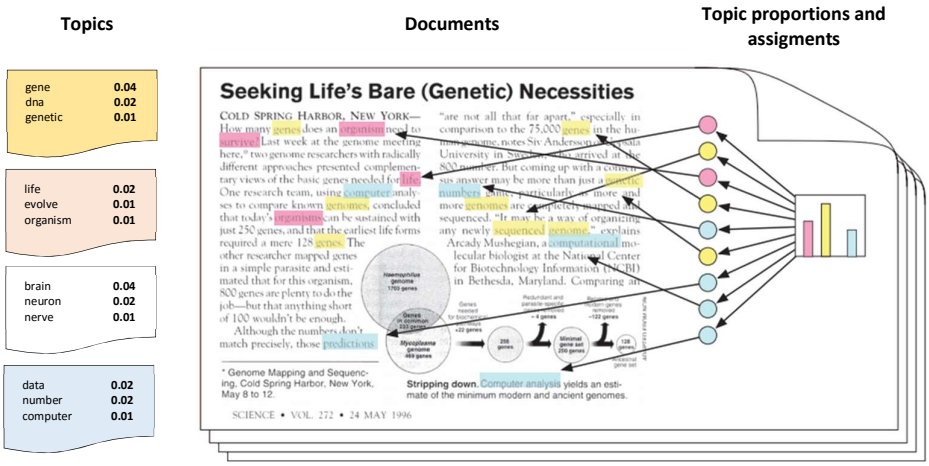

Ilustración 15: Funcionamiento de LDA dado un artículo

En la Ilustración 15, la distribución sobre tópicos colocaría probabilidades en genética, data análisis y biología evolutiva, y cada palabra sería extraída desde uno de esos tres tópicos. Note que el siguiente artículo en la colección puede ser sobre data análisis y neurociencia; su distribución sobre tópicos colocaría

probabilidades sobre esos dos tópicos. Ésta es la característica distintiva de latent Dirichlet allocation: todos los documentos en la colección comparten el mismo set de tópicos pero cada documento los exhibe en distinta proporción.

LDA y otros topic models son parte del amplio campo del modelamiento probabilístico. En el modelo probabilístico generativo, se trata los datos que surgen del proceso generativo que incluye variables ocultas. Este proceso generativo define una distribución de probabilidad conjunta sobre variables observadas y aleatorias ocultas. Lleva a cabo un análisis de datos mediante el uso de esa distribución conjunta para calcular la distribución condicional de las variables ocultas dadas las variables observadas. Esta distribución condicional también es conocida como distribución posterior.

LDA cae precisamente en este marco. Las variables observadas son las palabras de los documentos; las variables ocultas son la estructura tópico; y el proceso generativo es como se describe aquí. El problema computacional de inferir la estructura oculta tema de los documentos es el problema de calcular la distribución posterior, la distribución condicional de las variables ocultas dados los documentos.

Podemos describir LDA más formalmente con la siguiente notación. Los tópicos son $\beta_{1:k}$, donde cada β_k es una distribución sobre el vocabulario (las distribuciones sobre palabras a la izquierda en la Ilustración 15). Las proporciones de tópicos para el documento dth son θ_d, donde $\theta_{d,k}$ es la proporción tópicos del tópico k en el dth documento (histograma en la Ilustración 15). Las asignaciones de tópicos para el dth documento son z_d, donde $z_{d,n}$ es la asignación tópicos para la n-sima palabra en el documento d (la moneda de color en la Ilustración 15: Funcionamiento de LDA dado un artículo). Finalmente, las palabras observados para el documento son W_d, donde $W_{(d,n)}$ es la nth palabra en el documento d, que es un elemento del vocabulario fijo.

Con esta notación, el proceso generativo de LDA se corresponde con la siguiente distribución conjunta de las variables ocultas y observadas.

$$p(\beta_{1:K}, \theta_{1:D}, z_{1:D}, w_{1:D}) = \prod_{i=1}^{K} p(\beta_i) \prod_{d=1}^{D} p(\theta_d)(\prod_{n=1}^{N} p(z_{d,n}|\theta_d)p(w_{d,n}|\beta_{1:k}, z_{d,n}))$$

Debe tenerse en cuenta que esta distribución se especifica una serie de dependencias. Por ejemplo, la asignación de tópicos $z_{d,n}$ depende de las

proporciones de tópicos por documento θ_d. Otro ejemplo es, la palabra observada $w_{d,n}$ depende de la asignación de tópicos $z_{d,n}$ y todos los tópicos $\beta_{1:k}$.

Estas dependencias definen LDA. Ellas están codificadas en los supuestos estadísticos detrás del proceso generativo, en la forma matemática particular de la distribución conjunta, y en tercer lugar, en el modelo gráfico probabilístico para LDA. El modelo gráfico probabilístico proveen un lenguaje gráfico para describir familias de distribuciones de probabilidad. Estas tres representaciones son formas equivalentes de describir los supuestos probabilísticos tras LDA.

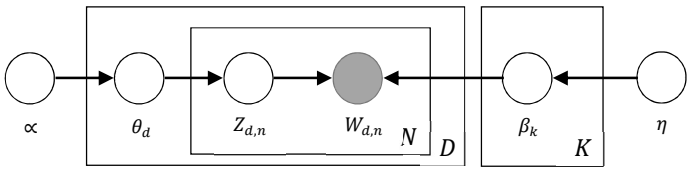

Ilustración 16: Modelo gráfico para LDA

La Ilustración 16 muestra el modelo gráfico para LDA. Cada nodo es una variable aleatoria y es etiquetado según su papel en el proceso generativo (ver Ilustración 15). Los nodos ocultos - las proporciones de tópicos, asignaciones y tópicos - no están sombreados. Los nodos observados - las palabras de los documentos – están sombreados. Los rectángulos (llamados placas) denotan replicación. La placa N denota la colección de palabras dentro de los documentos. La placa D denota la colección de documentos dentro de la colección [13].

Con Spark 1.3, MLlib ahora soporta LDA. Siendo el primer algoritmo MLlib construido sobre GraphX.

La implementación de LDA en MLlib, toma la colección de documentos como vectores de word counts. Aprende clustering utilizando el algoritmo probabilístico de esperanza-maximización, dando como resultado los tópicos inferidos y la distribución sobre estos tópicos. De acuerdo a los desarrolladores se escogió el algoritmo de esperanza maximización sobre otros enfoques, debido a que se converge a una solución en un menor número de iteraciones.

Además de inferir tópicos, LDA infiere la distribución sobre tópicos por cada documento de la colección, esta distribución puede ser usada de diversas formas

- **Clustering:** los tópicos son los centros de un clúster y los documentos están asociados a múltiples clústeres (tópicos). Este clustering puede ayudar a organizar o sumarizar colecciones de documentos.

- **Generación de valores:** LDA puede generar valores para ser usados por otros algoritmos de ML. La distribución sobre tópicos por cada documentos proporciona valores numéricos que pueden ser pasados como datos de entrada a otros algoritmos como Regresión Logística o Arboles de Decisión para tareas de predicción.

- **Reducción de dimensionalidad:** Cada distribución de documento sobre tópicos recoge un breve resumen del documento. La comparación entre documentos con esta característica reducida puede ser más significativa que la comparación entre los documentos originales.

En definitiva, su uso resulta intuitivo pues requiere de unos pocos parámetros, eso sí, los parámetros deben ser establecidos manteniendo un criterio conservador y de pensando siempre en la capacidad de computación con la que se cuente. Así finalmente, la configuración LDA utilizada fue la siguiente

```
…
   val documents: RDD[(Long, Vector)] =  tokenized…
…
   val numTopics = 5
   val lda = new LDA().setK(numTopics).setMaxIterations(5)
   val ldaModel = lda.run(documents)
…
```

Algoritmo 5: Implementación básica del modelo LDA en Spark

Básicamente son requeridos tres valores: los documentos, k: el número de tópicos requeridos y maxIterations: el número máximo de iteraciones a ejecutar, es necesario recordar que estos valores deben adecuarse a los datasets que se procesen.

4.6. Stage 5: Visualization

El tiempo que toma computar los datos es variable, tras lo cual resultaba necesario visualizar los datos en pantalla, por lo que se implementó una rutina sencilla para este propósito, limitando la visualización de los términos hallados a 10 por tópico, y mostrando el par de valores (termino, frecuencia), como se aprecia en el Algoritmo 6,

```
val topicIndices = ldaModel.describeTopics(maxTermsPerTopic = 10)
topicIndices.foreach { case (terms, termWeights) =>
  println("TOPIC:")
  terms.zip(termWeights).foreach { case (term, weight) =>
    println(s"${vocabArray(term.toInt)}\t$weight")
  }
  println()
```

Algoritmo 6: Visualización de resultados

5. Resultados

5.1. Experimentos

Durante el desarrollo de los experimentos las pruebas fueron realizadas en un ordenador portátil con microprocesador core i5-3337U de 1.80 Ghz, 6GB de memoria RAM y Sistema Operativo Ubuntu v14.04; El software empleado fue Apache Spark 1.3, Scala 2.11 y JDK 7; el clúster de Spark fue desplegado en modo Standalone, configurando 1 master y 5 workers.

Para las pruebas se utilizaron tres datasets los cuales fueron recopilados entre agosto y noviembre de 2014 en intervalos de tiempo distintos, mediante la suscripción a un determinado objeto Tag en la API de Instagram, De acuerdo a las especificaciones brindas en el capítulo 4, los documentos debían contener uno de los siguientes tags: #worldcup2014, #Desigual y #lavidaeschula, en el texto que acompaña a la imagen cuando es posteada; posteriormente los datos fueron clasificados de acuerdo a este criterio, la información relativa a los datos conseguidos se muestran en el siguiente cuadro:

	#desigual	#Lavidaeschula	#worldcup14
Posts procesados	3258	12471	501
Tamaño de bytes	8.4 MB	33.7 MB	995.1 kb
Nro de palabras	43763	185390	6594

5.2. Evaluación de performance

5.2.1. #wolrdcup2014 dataset

Debido a que el dataset del tag #worldcup2014 era el más pequeño el procesamiento de los datos fue el más rápido de todos pudiendo realizarse 3, 5 y 7 iteraciones, buscando 5 tópicos, estos datos la limitación de datos obviamente

limitaba la variedad de palabras, por lo que los resultados fueron algo limitados; además se aprecia duplicidad de términos en varios tópicos.

No obstante, puede apreciarse que en el tópico 5, referencias al partido disputado entre las selecciones de Holanda y México en octavos de final de la copa; el tópico 3 por su parte, hace referencia a la selección italiana y frases de aliento. A continuación se muestra el resultado del experimento:

```
TOPIC 1:
Forza          0.006397058726065184
partido        0.006308478001907601
after          0.00612401501266507
going          0.006089297081291318
more           0.005662605357545326
love           0.005173311595161192
netherlands    0.004942000624331346
fifa           0.004910273753898702
watching       0.004793733429093078
your           0.0047628314546245545


TOPIC 2:
forza          0.005192139298507153
after          0.004892488884851435
side           0.004873250064472125
first          0.004837548137749428
partido        0.004762856710768013
netherlands    0.004658646778099761
going          0.004562908515970665
more           0.004541787077373526
viva           0.004385200170224449
watching       0.00426131019197686

TOPIC 3:
partido        0.005336946774837449
number         0.005272593106634962
love           0.005058633457665034
fifa           0.004805553389683395
after          0.0047837458373728055
mundial        0.004618549288216553
forza          0.0045615949936217825
italy          0.0044705153252251025
viva           0.0043077650439136755
life           0.004281857319983886

TOPIC 4:
your           0.005547642607049927
still          0.004885895609066586
going          0.004780131524445958
good           0.0045066022642741045
italy          0.004323499430395682
first          0.004320382388216681
```

```
before          0.004276455543470316
partido         0.00422209749786537
after           0.004147115094885905
netherlands     0.004120741395620476

TOPIC 5:
netherlands     0.006129318790863058
mexico          0.004967228287134977
fifa            0.00479691658272246
brazil          0.004603665909379079
viva            0.004562711080117107
number          0.0045547182071077365
about           0.00450232633818077
going           0.004358884826396869
forza           0.004263219484812109
mundo           0.0042261413030245465
```

5.2.2. #Desigual dataset

El dataset correspondiente a Desigual, por su parte brindaba una cantidad de datos más aceptable, por lo que resultaba plausible obtener resultados coherentes, tras realizar las pruebas de rigor se apreció que las limitaciones de memoria del ordenador que se utilizaba conducían a fallo por saturación, lo que se pudo apreciar al asignar el número de iteraciones en 3 y 5, buscando 5 tópicos; por obvias razones se optó por reducir el número de iteraciones a 2 y se incrementaron el número de workers a 7; con esta configuración se estimaron 53 minutos en promedio para la obtención de resultados.

Los resultados obtenidos no guardan mayor coherencia entre los términos hallados, posiblemente debido al reducido número de iteraciones asignado, otro problema que se puede apreciar es la aparición de términos de uso común en habla inglesa de cuatro caracteres, lo que hacen que escapen al filtro establecido, y a la depuración de las stop words; otro detalle que llama fuertemente la atención es la aparición de términos en los tópicos inferidos no pertenecientes al habla inglesa o derivados de la lengua latina, probablemente se pueda afirmar que la marca desigual tiene una fuerte presencia en regiones de medio oriente y asia, ya que un análisis sobre las locaciones de donde procedían los post arrojaron varias incidencias sobre ciudades como: Bangkok (Thailandia), Bhaucha Dhakka (India) y Maldives (Mumbai). El resultado de las pruebas se muestra a continuación:

TOPIC 1:
ماشاءالله 0.004468097006780269
just 0.004131747790192916
like 0.003053790168881109
that 0.0027004647591390425
happy 0.002465762594539976
يسعدلي 0.0024515284197282714
best 0.00232131117124172
minha 0.0020576939420375354
todos 0.002042869711385993
الله 0.0019363494093312841

TOPIC 2:
just 0.003873094564637093
ماشاءالله (Allahmashaouallah) 0.0035151965955378043
that 0.0031203225212475772
like 0.0026153580197223417
التي (Which) 0.0025304246476429014
يسعدلي (Asaadla) 0.002401491738106858
التي (My heart) 0.0023665622039845953
الحنين (Nostalgia) 0.0019909746003912857
best 0.0019291148897819525
apenas 0.0019265543698189817

TOPIC 3:
ماشاءالله (Mashallah) 0.003037225437859836
that 0.0028397584789944755
just 0.00266532317366699
like 0.0024058402898522934
سعاده (Saadeh) 0.0018991767413521678
todos 0.0018570964469813535
يسعدلي (Asaadla) 0.0018346075323018775
الحنين (Nostalgia) 0.0018248317653375627
الله (God) 0.0017642109781828422
from 0.001720711593908249

TOPIC 4:
ماشاءالله (Mashallah) 0.003415802731232858
just 0.0028399771082024917
like 0.002801392786545508
that 0.0024385172036357535
التي (Which) 0.00238193975165711
يسعدلي (Asaadla) 0.0021320307523528016
best 0.002079974623212824
happy 0.002057300189488187
سعاده (happiness) 0.0018726287188348836
last 0.0018066621706080738

TOPIC 5:
ماشاءالله (Mashallah) 0.003583197140684714
يسعدلي (Asaadla) 0.0027622366392573944
like 0.002638772941161914
just 0.0025732906778885654
الأعزاء (Dear) 0.0024545379233554634
that 0.002410644834115247
minha 0.002389245631137365

متابعيني (Mtabaana)	0.0021237871243404613
سعاده (happiness)	0.0020168276581463518
from	0.001995939377015839

5.2.3. #lavidaeschula dataset

Finalmente, debido a los mismos motivos que produjeron problemas al procesar el dataset Desigual, el dataset la vida es chula no pudo ser computado pese a extenderse el número de tópicos a 10, reducirse el número de iteraciones a 1 y ampliar el número de workers a 10, originando sendos errores al superar las 2 horas de iniciada la computación

6. Conclusiones

6.1. Conclusiones

El performance de la aplicación fue influenciado en gran medida por limitaciones del hardware, no obstante también fue posible apreciar que ciertos parámetros de la implementación también influyen en la performance de la aplicación. En importante entonces, considerar factores como el tamaño del dataset (o la cantidad de palabras que contiene), para poder establecer parámetros idóneos en el modelo LDA, en este caso fueron los valores del número de iteraciones y el número de tópicos buscados los que debieron ser modificados para poder lograr una aproximación al resultado buscado.

Por otra parte, el algoritmo empleado en la etapa de pre-processing y filtering no presentaba necesidad de distinguir el idioma de los datos, al igual que el modelo LDA; de los resultados obtenidos en las pruebas se aprecian términos inferidos que no pertenecían a palabras de lenguas romances o anglosajonas, la utilidad de los términos hallados será relativa al tipo de análisis que se desee realizar, aunque definitivamente permiten conocer información relevante.

Durante el desarrollo de este trabajo fueron utilizados de forma directa e indirecta todos los componentes del Stack de Spark, considerando que el proyecto se encuentra en desarrollo, es posible afirmar que cumple con su propósito de forma eficiente; sin embargo el componente Spark Streaming aún no es estable del todo, ya que en repetidas oportunidades se han podido apreciar errores de funcionamiento durante distintas fases del procesamiento de datos, en una aplicación que generaba resultados funcionales durante otros intentos de ejecución, todo esto sin que se varíe ningún factor involucrado.

Los algoritmos disponibles en MLlib, solo podían implementarse en aplicaciones que trabajasen en batch, incluyendo el algoritmo LDA empleado en

el presente trabajo, lo que era una gran limitante para iniciativas que desearan trabajar con streams . Esto cambiaría tras el lanzamiento de la versión 1.3 de Spark en el mes de marzo, esta nueva versión puso a disposición el k-means streaming que pese a ser uno de los algoritmos de clusterización más básicos, su nueva funcionalidad puede dar mucho juego en el futuro; actualmente se vienen desarrollando implementaciones de igual índole con otros algoritmos.

6.2. Trabajo futuro

El presente trabajo sienta las bases para futuros trabajos en la misma línea pudiendo, por ejemplo, complementarse la fase inicial previa a la etapa de ingestión de datos, implementando el procesado de imágenes, cuyos resultantes podrían ser metadatos que interpreten la imagen procesada, pudiendo realizarse la clasificación posterior en base a estos estos datos. Los resultados obtenidos serian de gran interés.

Otra posibilidad en sería realizar el procesado de imágenes directamente en Spark, es decir ver la forma de ingresar directamente los archivos de imagen como streams, realizar la interpretación de los ficheros y realizar la clasificación en base a estos resultados.

Para explorar estas alternativas, se espera profundizar en el tema de Machine Learning e ir un paso más allá; Existe el convencimiento de que el acelerado ritmo de avances en Apache Spark y tal vez otras tecnologías, eventualmente permitirán la implementación de aplicaciones Deep Learning en streaming.

7. Bibliografía

[1] L. M. Aiello, G. Petkos, C. Martin, D. Corney, S. Papadopoulos, R. Skraba, A. Goker, Y. Kompatsiaris y J. Alejandro, «Sensing trending topics in Twitter,» *IEEE Transactions on Multimedia,,* vol. 15, nº 6, pp. 1268 - 1282, 2013.

[2] G. Barbier, Z. Feng, P. Gundecha y H. Liu, Provenance Data in Social Media, Chicago: Morgan & Claypool, 2013.

[3] L. Ingrid, «Tumblr Overtakes Instagram As Fastest-Growing Social Platform,» 25 11 2014. [En línea]. Available: http://techcrunch.com/2014/11/25/tumblr-overtakes-instagram-as-fastest-growing-social-platform-snapchat-is-the-fastest-growing-app/. [Último acceso: 19 12 2014].

[4] J. Bell, Machine Learning: Hands-On for Developers and Technical Professionals, Indianapolis: John Wiley & Sons, Inc., 2015.

[5] N. Sawant y H. Shah, Big Data Application Architecture Q & A, New York, NY: Apress Media, 2013.

[6] H. Karau, A. Konwinski, P. Wendell y M. Zaharia, Learning Spark: Lightning-Fast Big Data Analysis, CA: O'Reilly Media Inc., 2015.

[7] Apache Spark, «Spark Programming Guide,» 11 Setiembre 2014. [En línea]. Available: http://spark.apache.org/docs/1.1.0/programming-guide.html. [Último acceso: 13 Setiembre 2014].

[8] D. Macmillan y E. Dwonsin, «The Wall Street Journal.com,» 9 Octubre 2014. [En línea]. Available: http://www.wsj.com/articles/smile-marketing-firms-are-mining-your-selfies-1412882222. [Último acceso: 23 Noviembre 2014].

[9] R. Xin, M. Zaharia y others, «Shark: SQL and Rich Analytics at Scale,» EECS Department, University of California, Berkeley, California, 2012.

[10] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker y I. Stoica, «Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing,» de *9th USENIX Symposium on Networked Systems Design and Implementation*, San Jose, CA, 2012.

[11] Apache Spark, «Spark Streaming Programming Guide,» 11 Setiembre 2014. [En línea]. Available: http://spark.apache.org/docs/1.1.0/streaming-programming-guide.html. [Último acceso: 13 Setiembre 2014].

[12] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker y I. Stoica, «Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing on Large Clusters,» de *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, Berkeley, CA, 2012.

[13] D. Blei, «Probabilistic Topic Models,» *Communications of the ACM,* vol. 55, nº 4, p. 77–84, 2012.

[14] W. Fan, A. Bifet, Q. Yang y P. Yu, «Collapsed Gibbs Sampling for Latent Dirichlet Allocation on Spark,» *Journal of Machine Learning Research,* vol. 36, pp. 17-28, 2014.

[15] Y. Wang, Z. Xuemin , S. Zhenlong , Y. Hao , W. Lifeng , J. Zhihui , W. Liubin , G. Yang , J. Zeng, . Y. Qiang y L. Ching , «Towards Topic Modeling for Big Data,» *arXiv,* vol. 4402v1, p. 1405, 2014.

[16] S. Tamura, K. Tamura, H. Kitakami y K. Hirahara, «Clustering-based Burst-detection Algorithm for Web-image Document Stream on Social Media,» de *2012 IEEE International Conference on Systems, Man, and Cybernetics*, Seoul, 2012.

[17] N. Sawant y H. Shah, «Big Data Application Architecture,» de *Big Data Application Architecure Q&A - A problem-solution approach*, New York, Springer, 2013, pp. 9-28.

[18] M. G. Noll, «Integrating Kafka and Spark Streaming: Code Examples and State of the Game,» 1 Octubre 2014. [En línea]. Available: http://www.michael-noll.com/blog/2014/10/01/kafka-spark-streaming-integration-example-tutorial/. [Último acceso: 13 Diciembre 2014].

[19] D. Blei, A. Ng y M. Jordan, «Latent Dirichlet allocation,» *Journal of Machine Learning Research,* nº 3, p. 993–1022, 2003.

[20] «Instagram Developer Documentation,» [En línea]. Available: https://instagram.com/developer/. [Último acceso: 2014 11 02].

[21] D. Blei y J. Lafferty , «Topic Models,» de *Text Mining: Classification, Clustering, and Applications . Chapman & Hall/CRC Data Mining and Knowledge Discovery Series*, In A. Srivastava and M. Sahami, editors, 2009.

[22] K. Finley, «W.I.R.E.D.com,» 10 Octubre 2014. [En línea]. Available: http://www.wired.com/2014/10/startup-crunches-100-terabytes-data-record-23-minutes/. [Último acceso: 15 Octubre 2014].

[23] Apache Spark, «Machine Learning Library (MLlib) Programming Guide,» 11 Setiembre 2014. [En línea]. Available: http://spark.apache.org/docs/1.1.0/mllib-guide.html. [Último acceso: 15 Octubre 2014].

[24] Apache Spark, «GraphX Programming Guide,» 26 Noviembre 2014. [En línea]. Available: http://spark.apache.org/docs/1.1.1/graphx-programming-guide.html. [Último acceso: 15 Diciembre 2014].

[25] Apache Spark, «Spark Standalone Mode,» 5 Agosto 2014. [En línea]. Available: http://spark.apache.org/docs/1.1.0/spark-standalone.html. [Último acceso: 18 Agosto 2014].

[26] Apache Spark, «Spark Streaming + Kafka Integration Guide,» 26 Noviembre 2014. [En línea]. Available: http://spark.apache.org/docs/1.1.1/streaming-kafka-integration.html. [Último acceso: 5 Enero 2015].

[27] Apache.org, «parallel Latent Dirichlet Allocation (LDA) atop of spark in MLlib,» 28 Setiembre 2014. [En línea]. Available: https://issues.apache.org/jira/browse/SPARK-1405. [Último acceso: 15 Noviembre 2014].

[28] K. Ballou, «Real Time Streaming with Apache Spark,» [En línea]. Available: http://www.zdatainc.com/2014/08/real-time-streaming-apache-spark-streaming/. [Último acceso: 16 Diciembre 2014].

[29] Duke - Computer Science, «Data-Intensive Systems: Real-time Stream Processing,» [En línea]. Available: http://www.cs.duke.edu/~kmoses/cps516/dstream.html. [Último acceso: 22 Enero 2015].

[30] H. Edelson, «Spark Streaming with Kafka and Cassandra,» 9 Setiembre 2014. [En línea]. Available: http://helenaedelson.com/?p=991. [Último acceso: 5 Enero 2015].

[31] R. Ho, «Spark: Low Latency, Massively Parallel Processing Framework,» 1 Julio 2014. [En línea]. Available: http://java.dzone.com/articles/spark-low-latency-massively. [Último acceso: 21 Enero 2015].

[32] M. T. Jones, «Procese big data en tiempo real con Twitter Storm,» 11 02 2013. [En línea]. Available: http://www.ibm.com/developerworks/ssa/library/os-twitterstorm/. [Último acceso: 10 02 2015].

[33] A. Torres, «Con 300 millones de usuarios, Instagram ya supera a Twitter,» 2014 12 11. [En línea]. Available: http://www.lanacion.com.ar/1751249-con-300-millones-de-usuarios-instagram-ya-supera-a-twitter. [Último acceso: 2015 03 04].

[34] Z. Qiu, B. Wu, B. Wang, C. Shi y L. Yu, «Collapsed Gibbs Sampling for Latent Dirichlet Allocation on Spark,» *Journal Machine Learning Research,* vol. 36, pp. 17-28, 2014.