GESTIÓN DE DATOS EN EL NEGOCIO AUDIOVISUAL EN REAL TIME

TRABAJO FIN DE MÁSTER

Melisa Fernández Montero JUNIO / 2018

RESUMEN

Big data, término evolutivo que describe cualquier cantidad voluminosa de datos estructurados, semiestructurados y no estructurados que tienen el potencial de ser extraídos para su inmediato análisis para encontrar información oculta, patrones recurrentes, nuevas correlaciones, etc.

Cada día se habla más de este término, es una tendencia en aumento que ha llegado para quedarse; en absoluto se trata "de una moda". Este, ayuda a las organizaciones a aprovechar sus datos y utilizarlos para identificar nuevas oportunidades con la finalidad de obtener movimientos de negocios más inteligentes, operaciones más eficientes, mayores ganancias y clientes más felices. Poco a poco se está integrando en nuestras vidas, pasando de ser algo desconocido a una forma más de comprender el comportamiento humano y nuestra relación con el entorno.

En este proyecto pretendemos aprovechar todas esas oportunidades que ofrece Big Data y analizar los datos obtenidos tanto de histórico como en real-time de las valoraciones / visualizaciones / tweets de usuarios sobre películas bajo una plataforma similar a Netflix para estudiar su comportamiento, gustos y hábitos con el fin de poner en práctica parte de las herramientas vistas durante el Máster.

Índice de contenido

1. Introducción

Uno de los ejemplos que aparece siempre sobre cómo el big data puede mejorar los resultados de una empresa es el de Netflix. En sus inicios, era sólo una empresa que alquilaba películas en formato de DVD por correo; pero en 2009 decidieron empezar a usar un algoritmo para predecir los gustos de sus suscriptores. Esta decisión les permitió crecer de manera espectacular y convertirse en la gran amenaza para la supervivencia de la televisión.

La clave de su éxito está en cómo han usado la información para comprender a su nicho de mercado y cómo esto los ha puesto en una situación destacada. Podemos destacar los siguientes puntos concretos en los que Netflix ha trabajado:

- Personalización de los contenidos para llegar a más consumidores: el acceso al mismo no es igual para todo el mundo. Cada persona ve lo que su algoritmo cree que le interesa.
- Usar los datos y su análisis para prevenir pérdidas y tomar decisiones estratégicas: los datos les han permitido conocer mejor a su audiencia y crear contenidos en función de sus intereses.

Tomando como base el modelo de Netflix queremos aplicar estos puntos a nuestro proyecto. En los siguientes apartados se explican las principales características del mismo.

2. Descripción general del proyecto

Los objetivos principales del proyecto es poder responder a los siguientes puntos:

- Cómo usuario de la plataforma:
  - Disponer de un sistema de recomendaciones personalizado
  - Top Ten películas mejor valoradas por género
  - Top Ten películas mejor valoradas por director
- Como empresa:
  - Analizar los tweets de las películas que actualmente están en cartelera para conocer el Deep Sentiment Analysis y conocer si compensa incluirla a catálogo o no.
  - Día de la semana que más se consume
  - Franja horaria que más se consume
  - Media en la que una película es visualizada al año / mes / día
  - Películas con un índice de visualización inferior a X
  - Género mejor valorado
  - Director mejor valorado

Para poder responder a estos puntos nos hemos basado en soluciones tecnológicas innovadoras basadas en Big Data y Data Science.

El código fuente se ha subido al siguiente repositorio:
https://github.com/yensaiFM/PFM_Kschool

3. Fuentes de datos

En el proyecto utilizaremos varias entradas de datos:

- Histórico de datos proporcionado por Kaggle https://www.kaggle.com/rounakbanik/the-movies-dataset. Este dataset contiene información de 45.000 películas lanzadas antes de Julio del 2017; así como votaciones y promedio de votos. También dispone de 26 millones de calificaciones de 270.000 usuarios para las 45.000 películas; estas calificaciones están en una escala del 1 al 5. Este dataset es una recopilación de TheMovieDB y GroupLens.
- Para la generación de votaciones en tiempo real utilizaremos un productor en Kafka
- Utilización de la API de TheMovieDB. Esta API nos permite consultar las películas que actualmente están en cartelera para posteriormente filtrar los tweets por estos hastag.
- Utilización de la API de Twitter en Streaming. Esta API nos permite consultar los tweets publicados sobre las películas que actualmente están en cartelera.

4. Arquitectura Lambda

La Arquitectura Lambda surge en los últimos años con el objetivo de dar respuesta global a diferentes necesidades de análisis Big Data: volumen de datos cada vez mayor, la necesidad de analizarlos y obtener valor de ellos lo antes posible. Intentaba proporcionar un sistema robusto tolerante a fallos, tanto humanos como de hardware, que fuera linealmente escalable y que permitiese realizar escrituras y lecturas con baja latencia.

La arquitectura propuesta para el proyecto se basa en la arquitectura Lambda. Esta arquitectura se basa en tres capas: la capa por lotes (batch layer), la capa de servicio (serving layer) y la capa de velocidad (speed layer).

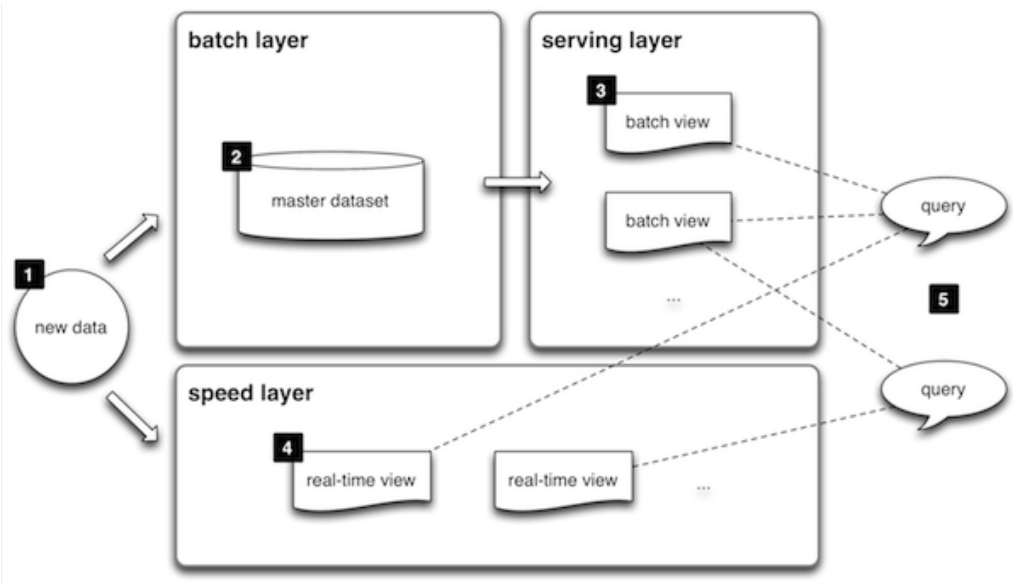


Ilustración 1 – Arquitectura Lambda

- Capa por lotes (batch layer) sus principales funciones son:
  - Gestionar el conjunto de datos maestro (master dataset), conjunto de datos inmutable y en continuo crecimiento; los datos nuevos nuevos se añaden a los ya existentes.
  - Precalcular las consultas por lotes obteniendo los resultados de la consulta; son las denominadas vistas por lotes (batch views).
- Capa de servicio (service layer): en esta capa se almacenan las vistas por lotes (batch views) y se indexan para que puedan ser consultadas con baja latencia.
- Capa de velocidad (speed layer): esta capa se encarga de procesar sólo los nuevos datos en tiempo real. Aquí también se generan vistas, denominadas vistas en tiempo real (realtime views), pero estas no se generan por lotes sino que se van actualizando según se van recibiendo nuevos datos. Esto compensa la latencia de las escrituras. Una vez que el procesamiento por lotes haya terminado de precalcular y las vistas por lotes se hayan actualizado, los resultados de las vistas en tiempo real ya no serán necesarios, por lo que se descartan.

Los pasos que se siguen son los siguientes:
- Los nuevos datos se envían a la capa por lotes y a la capa de velocidad.
- En la capa por lotes, los nuevos datos son añadidos al conjunto de datos maestro y se precalculan las consultas generándose las vistas por lotes.
- Las vistas por lotes se almacenan en la capa de servicio y se indexan para que puedan ser consultadas.
- En la capa de velocidad, los nuevos datos van actualizando los vistas en tiempo real.
- Cuando se realiza una consulta, esta se resuelve obteniendo los resultados de las vistas por lotes y las vistas en tiempo real, se unen o combinan sus resultados.

5. Arquitectura del proyecto

La siguiente ilustración muestra la arquitectura propuesta para el proyecto; donde podemos observar las distintas capas comentadas anteriormente.

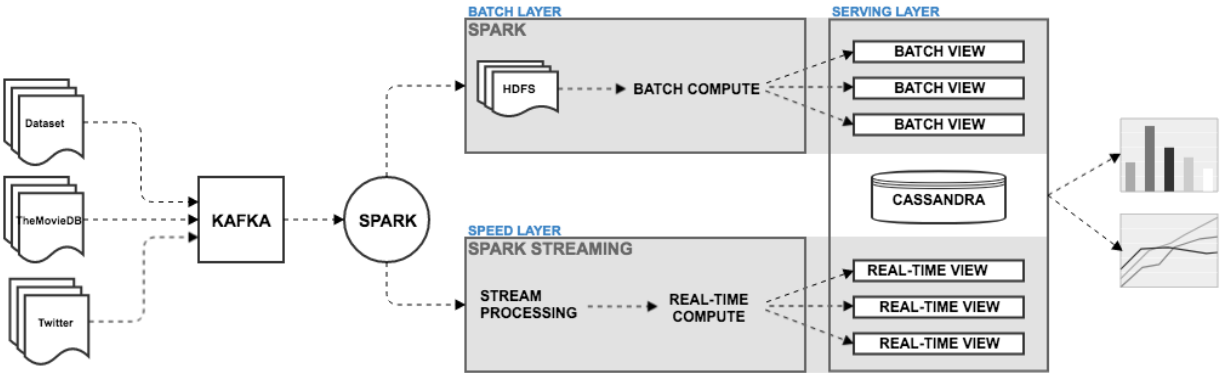


Ilustración 2 – Arquitectura Lambda

- En primer lugar, están las fuentes de datos (Data Sources) que representan todo tipo de fuentes de información disponibles: históricos de películas y valoraciones, información en real time de valoraciones, información en real time de los tweets sobre películas.
- Estos datos pasan al bloque de ingestión de datos (Data Ingestion Block), aquí los datos son almacenados en una cola, se realizan tareas de limpieza y se serializan en POJOs. Estos datos, serán enviados tanto a la capa por lotes (batch layer) como a la capa de velocidad (speed layer). La herramienta utilizada para realizar esta ingesta de información es Apache Kafka; es un sistema de almacenamiento publicador / subscriptor distribuido, particionado y replicado.
- La capa por lotes (batch layer) se encargará de:
  - Almacenamiento de datos (Data Storage Layer). Estos datos se denominan conjunto de datos maestro (master dataset), son inmutables y sólo permite añadir datos. Utilizaremos HDFS, sistema de archivos distribuidos tolerantes a fallos y auto reparable que permite de forma transparente almacenar un archivo de datos en diferentes máquinas de un cluster de Hadoop.
  - Procesamiento de datos (Data Processing Layer). Se itera continuamente sobre el conjunto de datos maestro recalculando las vistas por lotes (batch views). La herramientas utilizadas para realizar este procesamiento de datos son YARN y Apache Spark. YARN es uno de los módulos básicos de Apache Hadoop utilizado para la programación de tareas y gestión de recursos. Apache Spark especialmente desarrollado para el procesado de datos masivos desarrollado en torno a la velocidad, facilidad de uso y análisis sofisticado.
  - Capa de acceso de datos (Data Access Layer), que permite el acceso, recuperación y análisis rápido de la información invocando a las vistas por lotes. Utilizaremos el componente Spark SQL proporcionado por Spark; este nos permite la ejecución de querys sql sobre los datasets distribuidos en memoria (RDDs).
- La capa de velocidad (speed layer) es la encargada de actualizar continuamente las vistas en tiempo real (realtime views) con los datos más recientes. Utilizaremos el componente Spark Streaming proporcionado por Spark; este nos ofrece soporte para procesamiento en casi tiempo real a través de un sistema de empaquetamiento de pequeños lotes.
- La capa de servicios (service layer) es la encargada de combinar los resultados de las vistas por lotes y las vistas en tiempo real. Esta salida puede ser utilizada por distintas aplicaciones. Este bloque es básicamente una base de datos distribuida para almacenar las vistas procesadas y permitir que se pueda acceder a las mismas para consulta y visualización. Utilizaremos la bbdd noSQL Cassandra que nos permite almacenar grandes cantidades de datos y proporciona alta disponibilidad.

1. Herramientas utilizadas

En esta sección se realizará un análisis de las herramientas utilizadas durante el desarrollo de este proyecto. De cada tecnología describiremos sus características principales.

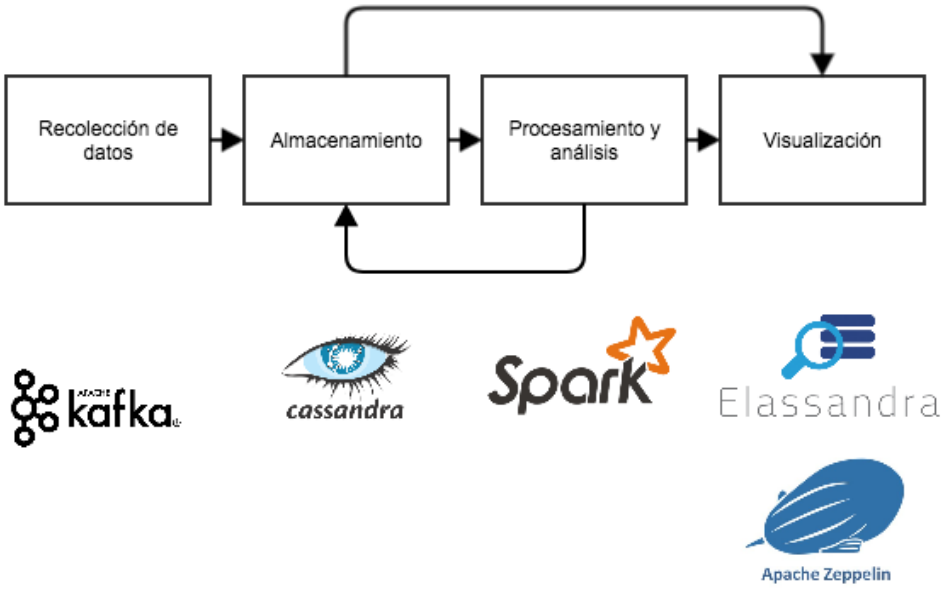


Ilustración 3 – Fases

1. Apache Hadoop

Framework de código abierto que permite el procesamiento distribuido de grandes conjuntos de datos en varios clústers. Hadoop separa y distribuye automáticamente los archivos que contienen los datos, ademas de dividir el trabajo en tareas más pequeñas y ejecutarlas de manera distribuida y recuperarse de posibles fallos automáticamente y de forma transparente al usuario.

Las principales funcionalidades de Hadoop son el procesamiento distribuido y el almacenamiento distribuido:

- Procesamiento distribuido: permite procesar grandes conjuntos de datos a través de clústers de computadoras, permitiendo a las aplicaciones trabajar con miles de nodos y petabytes de datos. Implementa un paradigma computacional denominado MapReduce, donde la aplicación se divide en muchos pequeños fragmentos de trabajo que se pueden ejecutar en cualquier nodo del clúster.
- Almacenamiento distribuido (HDFS – Hadoop Distributed File System): consiste en un sistema de ficheros distribuido ya que estos no se almacenan en una única máquina, sino que su información se distribuye en distintos nodos del clúster.

Incluye los siguientes módulos básicos:

- Hadoop Common: proporciona el acceso a los sistemas de archivos soportados por Hadoop y contiene el código necesario para poder ejecutar el framework.
- Hadoop Distributed File System (HDFS): sistema de ficheros distribuido; se encuentra optimizado para grandes flujos y trabajar con ficheros grandes en sus lecturas y escrituras. Su diseño reduce la E/S en la red. La escalabilidad y disponibilidad son otras de sus claves, gracias a la replicación de los datos y tolerancia a los fallos.
- Hadoop YARN (Yet Another Resource Negotiator): es una tecnología de gestión de clústers. Se compone de un gestor de recursos central y un gestor por cada nodo, que se ocupa de controlar un único nodo.
- Hadoop MapReduce: sistema basado en YARN para el procesamiento paralelo de grandes conjuntos de datos.

2. Apache Kafka

Sistema de mensajería distribuido, particionado y replicado basado en el patrón publish/subscribe. Sus tres características principales son las siguientes:

- Permite publicar y suscribirse a flujos de registros de forma similar a una cola de mensajes.
- Es tolerante a fallos.
- Permite procesar los flujos de registros a medida que ocurren.

Centrándonos en la primera característica, Kafka se ejecuta como un clúster en uno o más servidores, los cuales almacenan los registros (mensajes) en colas (topics). Cada mensaje consta de una clave, una marca temporal y un valor.

Dispone de 4 APIs:

- API de Producer: permite que una aplicación publique un mensaje a uno o más topics de Kafka.
- API de Consumer: permite que aplicación se suscriba a uno o más topics y procese el flujo de mensajes que previamente los Producers han creado en esos topics.
- API de Streams: permite que una aplicación actúe como un procesador de streams, consumiendo una secuencia de entrada de uno o más topics y produciendo una secuencia de salida a uno o más topics de salida, transformando efectivamente los streams de entrada a los flujos de salida.
- API de Connector: permite crear y ejecutar productores reutilizables o consumidores que conectan topics de Kafka a aplicaciones o sistemas de datos existentes.

Teniendo en cuenta las dos primeras APIs tenemos una serie de procesos, Producers, que generan mensajes que se organizan en topics dentro de los brokers. Estos brokers publican sus colas para que otros procesos, Consumers, los procesen. Kafka tiene una dependencia de Apache Zookeeper; este ofrece un servicio para la coordinación de procesos distribuidos altamente confiable que da soluciones a varios problemas de coordinación de grandes sistemas distribuidos.

3. Apache Spark

Framework de computación de clúster de código abierto; está escrito en el lenguaje de programción Scala y se ejecuta en el entorno JVM. Proporciona una API de programación para Scala, Java, Python y R.

Spark es un motor rápido y general para procesamiento de datos a gran escala. Proporciona una interfaz para programar clústeres enteros con paralelismo implícito y tolerancia a fallos.

Una de las ventajas que hace a Spark un sistema eficiente y rápido para el tratamiento de datos es que los RDD se almacenan en memoria, permitiendo un acceso más rápido a ellos para la realización de operaciones complejas.

Algunas de las ventajas más notables son:

- Procesamiento en memoria: la característica principal de Spark es que tiene la capacidad de realizar procesamiento en memoria dentro de un clúster, aumentando la velocidad de procesamiento. La computación en memoria significa almacenar los datos en RAM y procesarlos en paralelo. Tb dispone de un motor de ejecución avanzado que facilita el cálculo en memoria.
- Velocidad: Hasta 100 veces más rápido que Hadoop MapReduce en memoria y 10 veces más rápido en disco.
- Tolerancia a fallos: diseñado para manejar de forma transparente los fallos de cualquier nodo del trabajo en el clúster, evitando así la pérdida de datos e información.
- Procesamiento de flujos de datos en tiempo real: es capaz de trabajar con datos en tiempo real.
- Compatible con Hadoop: puede ejecutarse dentro en la primera generación de Hadoop sobre MapReduce, o sobre YARN en la segunda generación de Hadoop.
- Evaluación perezosa (lazy evaluation): cuando se llama a una operación, no se ejecuta inmediatamente; mantiene el registro de esa operación y no la ejecutará hasta que no sea necesaria.
- Fácil de usar: permite escribir aplicaciones en Java, Scala, Python y R.
- Módulos: dispone de bibliotecas adicionales que son parte del ecosistema y proporcionan capacidades adicionales como analítica Big Data y áreas de aprendizaje automático.

Los módulos centrales del sistema son:

- Spark Core Engine: ofrece la funcionalidad para el procesado de la información distribuida en memoria. La base del api son los datasets distribuidos en memoria (RDDs) sobre los que se puede realizar una serie de acciones y transformaciones MapReduce.
- Spark SQL: permite la ejecución de querys sql sobre los datasets distribuidos en memoria (RDDs), internamente se transforman esas consultas a trabajos en memoria implementados con MapReduce.
- Spark Streaming: ofrece funcionalidad para el procesado de la información en tiempo real, para ello hace uso de conectores a fuentes de datos como Twitter, Kafka, etc y procesa la información en ventanas temporales de datos. La manera en que Spark incorpora el procesado de flujos de

datos sin la necesidad de rehacer todos los demás componentes de la plataforma consiste en capturar pequeños paquetes de datos del flujo de entrada cada cierto intervalo de tiempo y presentarlos como un RDD. De este modo, el flujo de entrada toma la forma de una serie continua de RDDs, lo que se denomina Dstream (Discretized Stream), el concepto sobre el que gira todo Spark Streaming.
- Spark Mlib: proporciona una serie de algoritmos machine learning implementados con Spark que se ejecutan sobre datasets distribuidos en memoria (RDDs).
- Spark Dataframe: proporciona una estructura al estilo de los dataframes de Python pero en memoria, ofrece mejor rendimiento que los RDDs y encapsula la funcionalidad MapReduce en funciones.
- Spark GraphX: es un motor para el análisis de grafos, construido sobre el núcleo de Spark y que permite a los usuarios crear, transformar y obtener conclusiones sobre datos estructurados.

Además de estos módulos, hay otros conectores desarrollados por terceros:
- Adaptadores de integración con otros productos como Cassandra (Cassandra Spark Connector).
- Existen adaptadores para el manejo de distintos tipos de fichero como puede ser Apache Parquet, Apache Avro, etc.

4. Cassandra

Base de datos NoSQL columnar distribuida basada en software libre y desarrollada por Facebook, permite alta escalabilidad a lo largo de distintos data centers sobre hardware estándar, ofrece alta disponibilidad sin que ningún nodo actúe como punto de fallo único.

La arquitectura de Cassandra es responsable de su capacidad para escalar, ejecutar y ofrecer ejecución de forma permanente en el tiempo. En lugar de utilizar un modelo esclavo-servidor o una arquitectura fragmentada difícil de mantener tiene un diseño en forma de anillo sin maestro siendo más fácil de instalar y mantener.

Todos los nodos tienen un papel idéntico; no existe el concepto de un nodo principal, todos los nodos se comunican entre sí. Esta arquitectura permite que sea capaz de manejar grandes cantidades de datos, miles de usuarios y miles de operaciones por segundo concurrentes incluso a través de múltiples data centers. No tiene ningún punto único de fallo ofreciendo disponibilidad y acceso continuo.
En caso de necesitar escalar únicamente hay que añadir nuevos nodos al clúster existente sin tener que parar el servicio.

Sus características principales son:
- Modelo descentralizado: todos los nodos tienen el mismo rol, por lo que cualquier nodo puede responder ante cualquier solicitud.
- Escalabilidad: las operaciones de lectura y escritura aumentan a medida que se añaden nuevos nodos.
- Capacidad de réplica y escalabilidad: un clúster puede ser replicado en distintos data centers a modo de réplica y cada clúster puede aumentar el número de nodos para aumentar la capacidad del mismo.
- Tolerancia a fallos: la información es replicada en distintos nodos para tolerancia a fallos y la gestión de errores es gestionada de forma transparente.
- Ofrece soporte para MapReduce, Apache Pig y Apache Hive: posee una integración con Apache Hadoop, por lo que ofrece soporte para MapReduce, Apache Pig y Apache Hive.
- Lenguaje CQL (Cassandra Query Language): posee un lenguaje propio de consulta, similar a SQL.

5. Apache Zeppelin

Es una implementación del concepto de web notebook, centrado en la analítica de datos interactivo mediante lenguajes y tecnologías como Shell, Spark, SparkSQL, Hive, Elasticsearch, R, …
El concepto de "notebook" fue introducido por iPython, que permitía trabajar sobre una interfaz web en lugar de sobre una shell. Este permite compartir los procesos con otros, de modo que pueden ser modificados y adaptados a sus necesidades.

Ventajas que ofrece:
- Máxima simplicidad: permite de una forma muy intuitiva manejar conexiones de datos rápidas con diversas fuentes de datos.
- Agnóstico del lenguaje: existen plugins o interpreter que permiten ampliar nuestra solución.
- Interfaz sobre Bootstrap y AngularJS: fácil de usar e intuitivo.

6. Implementación

Este apartado muestra detalles concretos de la implementación llevada a cabo para este proyecto. Con el fin de que sea más sencillo entenderlo, este capítulo se dividirá en los distintos flujos de datos que serán procesados: datos de películas, datos de reviews, datos de tweets. De esta forma veremos de forma independiente todo el flujo que sigue.

En este desarrollo para tocar varios lenguajes de programación se decidió implementar la funcionalidad de Kafka con Java y la funcionalidad de Spark con scala.

1. Datos de películas

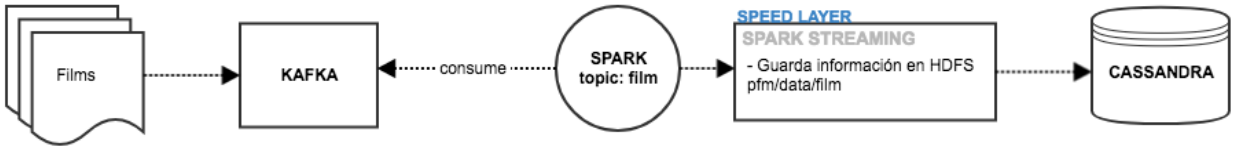
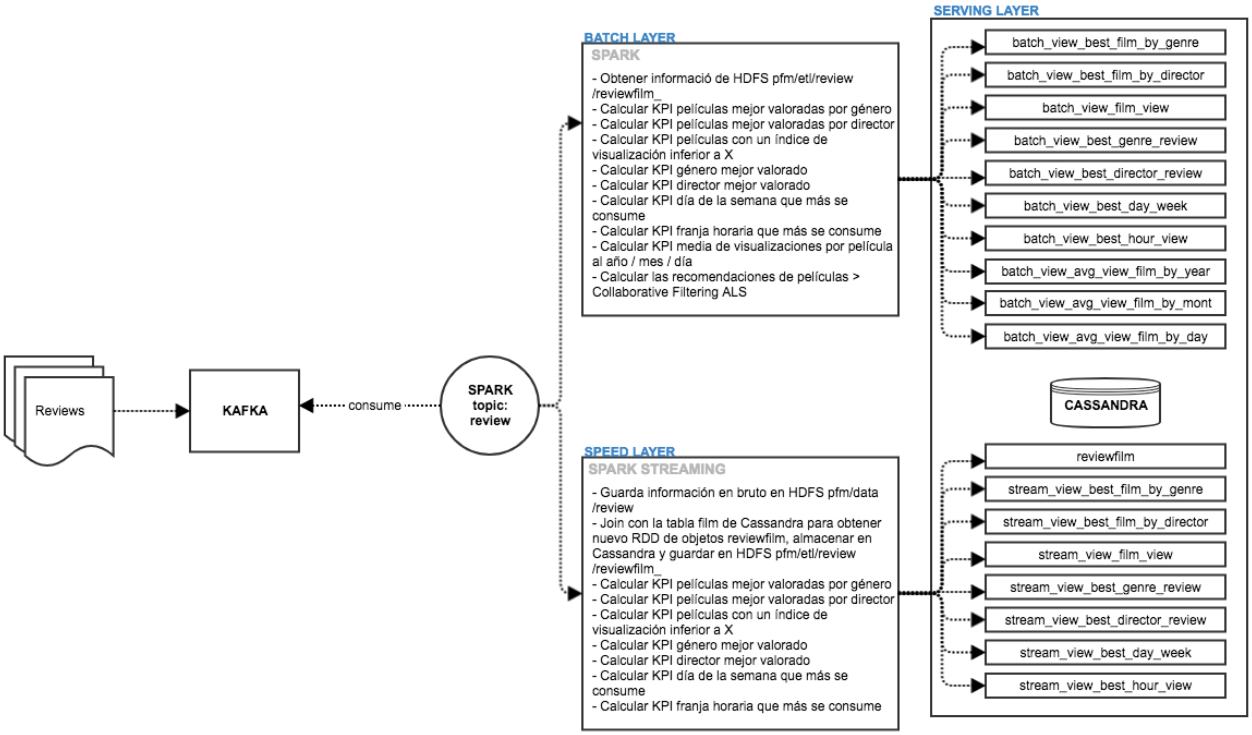


Ilustración 4 – Flujo que siguen los datos de películas

En la siguiente ilustración podemos observar el flujo que siguen los datos de películas dentro de la aplicación; a continuación lo detallaremos:

1. Datos, existen dos posibles entradas:
    1. Dataset, como necesitamos una primera carga para tener información utilizaremos el dataset de Kaggle comentado anteriormente.
    2. La API de TheMovieDB nos proporcionará información de nuevas películas en real-time.
2. Capa de recolección de datos, en Kafka se ha realizado:
    1. Creación del topic film: bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic film --partitions 1 --replication-factor 1
    2. Se ha creado el productor ProducerFilm; este obtiene la información en formato JSON y la serializa en la clase film. Ha sido necesario crear el serializador y deserializador correspondiente.
3. Capa de procesamiento de datos, en Spark Streaming se ha realizado:
    1. Para que Spark Streaming pueda consumir de Kafka se ha metido la dependencia al conector spark-streaming-kafka-0.8_2.11
    2. Para que Spark pueda almacenar la información en Cassandra se ha metido la dependencia al conector spark-cassandra-connector
    3. Se ha creado el objeto StreamingFilmJob que será el encargado de:
        1. Consumir el topic film de kafka
        2. Guardar la información en bruto en HDFS en el directorio hdfs://localhost:9000/pfm/data/film/
        3. Almacenar la información en la tabla pfm.film de Cassandra.
4. Capa de almacenamiento, en Cassandra hemos creado el keyspace pfm y la tabla film:
    1. CREATE KEYSPACE pfm WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 1 };
    2. CREATE TABLE film(movieId int PRIMARY KEY, imdbId text, title text, adult boolean, genres text, originallanguage text, productioncompanies text, releasedate text, runtime int, status text, director text);

2. Datos de reviews



En la siguiente ilustración podemos observar el flujo que siguen los datos de valoraciones dentro de la aplicación; a continuación lo detallaremos:

1. Datos, existen dos posibles entradas:
    1. Dataset, como necesitamos una primera carga para tener información utilizaremos el dataset de Kaggle comentado anteriormente.
    2. Dispondremos de un productor que irá enviando nuevas valoraciones cada cierto tiempo para simular el real time.
2. Capa de recolección de datos, en Kafka se ha realizado:
    1. Creación del topic review: bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic review --partitions 1 --replication-factor 1
    2. Se ha creado el productor ProducerReview; este obtiene la información en formato csv y la serializa en la clase review. Ha sido necesario crear el serializador y deserializador correspondiente.
3. Capa de procesamiento de datos, en Spark Streaming se ha realizado:
    1. Para que Spark Streaming pueda consumir de Kafka se ha metido la dependencia al conector spark-streaming-kafka-0.8_2.11
    2. Para que Spark pueda almacenar la información en Cassandra se ha metido la dependencia al conector spark-cassandra-connector
    3. Se ha creado el objeto StreamingReviewJob que será el encargado de:
        1. Consumir el topic review de kafka
        2. Guardar la información en bruto en HDFS en el directorio

hdfs://localhost:9000/pfm/data/review/

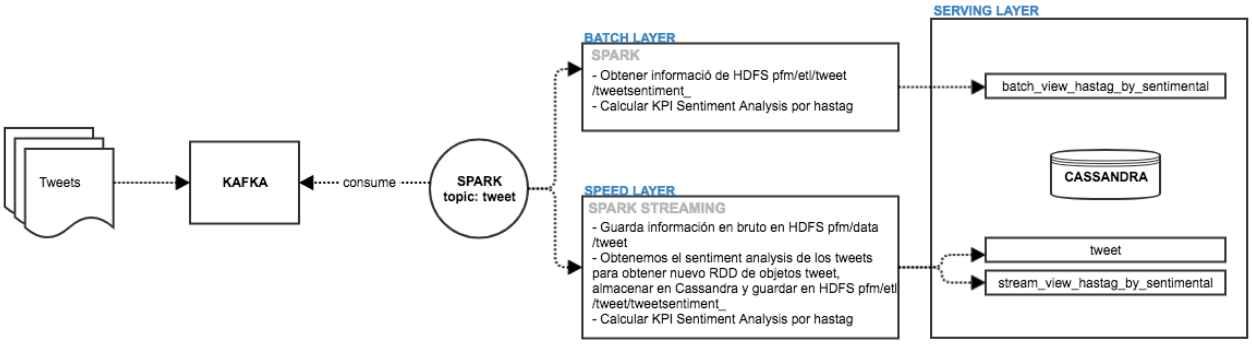
3. Para tener la relación entre la valoración y la película correspondiente crearemos un nuevo RDD juntando la información del RDD de reviews y la tabla pfm.film de Cassandra; para ello utilizaremos la funcionalidad joinWithCassandraTable. Este nuevo RDD se almacenará en la tabla pfm.reviewfilm de Cassandra.
4. Guardamos este nuevo RDD en HDFS en el directorio hdfs://localhost:9000/pfm/etl/review/reviewfilm_ en formato Parquet para que sea consumido posteriormente por el proceso Batch. Parquet es un formato de almacenamiento en columnas de código abierto para Hadoop. Soporta esquemas de codificación y compresión eficientes.
5. Procedemos al calculo de los distintos KPIs. En este punto como los cálculos se hacen por cada Dstream es necesario validar antes de guardar el cálculo en la view si ese row existe previamente para actualizar en lugar de insertar. Los KPIs que hemos calculado son los siguientes:
    1. Películas mejor valoradas por género: se define la clave (movieid, genres) se agrupa por dicha clave y se suman las valoraciones. Los resultados se almacenan en la view "stream_view_best_film_by_genre".
    2. Películas mejor valoradas por director: se define la clave (movieid, director) se agrupa por dicha clave y se suman las valoraciones. Los resultados se almacenan en la view "stream_view_best_film_by_director".
    3. Películas con un indice de visualización inferior a X: aquí se ha definido que una valoración equivale a una visualización. Se define la clave (movieid) se agrupa por dicha clave y se hace un conteo. Los resultados se almacenan en la view "stream_view_film_view".
    4. Género mejor valorado: Se define la clave (genres) se agrupa por dicha clave y se suman las valoraciones. Los resultados se almacenan en la view "stream_view_best_genre_review".
    5. Director mejor valorado: Se define la clave (director) se agrupa por dicha clave y se suman las valoraciones. Los resultados se almacenan en la view "stream_view_best_director_review".
    6. Día de la semana que más se consume: Se define la clave (dayOfWeek, month, year) se agrupa por dicha clave y se hace un conteo. Los resultados se almacenan en la view "stream_view_best_day_week".
    7. Franja horaria que más se consume: Se define la clave (day, month, year, hour) se agrupa por dicha clave y se hace un conteo. Los resultados se almacenan en la view "stream_view_best_hour_view".
6. Capa de almacenamiento, como el keyspace ya se había creado anteriormente únicamente se han creado las views:
    1. CREATE TABLE reviewfilm(movieId int PRIMARY KEY, userId int, rating double, timestamp timestamp, director text, genres text, title text);
    2. CREATE TABLE stream_view_best_film_by_genre(movieid int, genres text, rating double, title text, PRIMARY KEY(genres, movieid));
    3. CREATE TABLE stream_view_best_film_by_director(movieid int, director text, rating double, title text, PRIMARY KEY(director, movieid));
    4. CREATE TABLE stream_view_film_view(movieid int, total int, title text, PRIMARY KEY(movieid));
    5. CREATE TABLE stream_view_best_genre_review(genres text, rating double, PRIMARY KEY(genres));
    6. CREATE TABLE stream_view_best_director_review(director text, rating double, PRIMARY KEY(director));
    7. CREATE TABLE stream_view_best_day_week(day text , month text, year int, total int, PRIMARY KEY(day, month, year));
    8. CREATE TABLE stream_view_best_hour_view(day int, month int, year int, hour int, total int, PRIMARY KEY(day, month, year, hour));

4. Capa de procesamiento de datos, en Spark (batch layer) se ha realizado:
    1. Utilizando la API de Spark SQL obtenemos la información del directorio HDFS hdfs://localhost:9000/pfm/etl/review/reviewfilm_
    2. Procedemos al calculo de los distintos KPIs; como los cálculos se hacen sobre todos los datos aquí no se tiene en cuenta si existe o no. Los KPIs que hemos calculado son los siguientes:
        1. Películas mejor valoradas por género: aplicamos la consulta SELECT movieid, genres, sum(rating) as rating, title FROM reviewfilm GROUP BY movieid, title, genres. Los resultados se almacenan en la view "batch_view_best_film_by_genre".
        2. Películas mejor valoradas por director: aplicamos la consulta SELECT movieid, director, sum(rating) as rating, title FROM reviewfilm GROUP BY movieid, title,

director. Los resultados se almacenan en la view "batch_view_best_film_by_director".
3. Películas con un indice de visualización inferior a X: aplicamos la consulta SELECT movieid, count(*) as total, title FROM reviewfilm GROUP BY movieid, title. . Los resultados se almacenan en la view "batch_view_film_view".
4. Género mejor valorado: aplicamos la consulta SELECT genres, sum(rating) as rating FROM reviewfilm GROUP BY genres. Los resultados se almacenan en la view "batch_view_best_genre_review".
5. Director mejor valorado: aplicamos la consulta SELECT director, sum(rating) as rating FROM reviewfilm GROUP BY director. Los resultados se almacenan en la view "batcj_view_best_director_review".
6. Día de la semana que más se consume: aplicamos la consulta SELECT date_format(CAST(timestamp/1000 as timestamp), 'EEEE') as day, date_format(CAST(timestamp/1000 as timestamp), 'MMMM') as month,date_format(CAST(timestamp/1000 as timestamp), 'YYYY') as year, count(*) as total FROM reviewfilm GROUP BY date_format(CAST(timestamp/1000 as timestamp), 'EEEE'), date_format(CAST(timestamp/1000 as timestamp), 'MMMM'), date_format(CAST(timestamp/1000 as timestamp), 'YYYY'). Los resultados se almacenan en la view "batch_view_best_day_week".
7. Franja horaria que más se consume: aplicamos la consulta SELECT day(CAST(timestamp/1000 as timestamp)) as day, month(CAST(timestamp/1000 as timestamp)) as month, year(CAST(timestamp/1000 as timestamp)) as year, hour(CAST(timestamp/1000 as timestamp)) as hour, count(*) as total FROM reviewfilm GROUP BY day(CAST(timestamp/1000 as timestamp)), month(CAST(timestamp/1000 as timestamp)), year(CAST(timestamp/1000 as timestamp)), hour(CAST(timestamp/1000 as timestamp)). Los resultados se almacenan en la view "batch_view_best_hour_view".
8. Media de visualizaciones por película al año / mes / día: aquí hemos tenido que realizar dos consultas por KPI, la primera para obtener el número de visualizaciones por película agrupada por año / mes / día y otra para obtener el total de visualizaciones por año / mes / día para poder obtener la media. Los resultados se almacenan en las views "batch_view_avg_view_film_by_year", "batch_view_avg_film_by_month" y "batch_view_avg_view_film_by_day".
9. Recomendaciones de películas según los gustos de los usuarios: en este punto hemos creado un objeto RecommendationMovies dónde almacenamos el modelo de Collaborative Filtering > Recommendation. Aquí hemos entrenado el modelo con los datos almacenados en HDFS hdfs://localhost:9000/pfm/etl/review/reviewfilm_ una vez entrenado y obtenido el valor de error (0.6562) se ha almacenado dicho modelo en hdfs://localhost:9000/pfm/model/recommendationALS.model para ser utilizado posteriormente en el proceso batch. Los resultados se almacenan en la view "batch_view_recommendation_films".

3. Capa de almacenamiento, como el keyspace ya se había creado anteriormente únicamente se han creado las views:
   1. CREATE TABLE batch_view_best_film_by_genre(movieid int, genres text, rating double, title text, PRIMARY KEY (genres, movieid));
   2. CREATE TABLE batch_view_best_film_by_director(movieid int, director text, rating double, title text, PRIMARY KEY (director, movieid));
   3. CREATE TABLE batch_view_film_view(movieid int, total int, title text, PRIMARY KEY(movieid));
   4. CREATE TABLE batch_view_best_genre_review(genres text, rating double, PRIMARY KEY(genres));
   5. CREATE TABLE batch_view_best_director_review(director text, rating double, PRIMARY KEY(director));
   6. CREATE TABLE batch_view_best_day_week(day text , month text, year int, total int, PRIMARY KEY(day, month, year));
   7. CREATE TABLE batch_view_best_hour_view(day int, month int, year int, hour int, total int, PRIMARY KEY(day, month, year, hour));
   8. CREATE TABLE batch_view_hastag_by_sentimental(hastag text, sentimental text, total int, PRIMARY KEY(hastag, sentimental));
   9. CREATE TABLE batch_view_avg_view_film_by_year(year int, movieid int, title text, avg float, PRIMARY KEY(year, movieid));
   10. CREATE TABLE batch_view_avg_view_film_by_month(year int, month int, movieid int, title text, avg float, PRIMARY KEY(year, month, movieid));
   11. CREATE TABLE batch_view_avg_view_film_by_day(year int, month int, day int, movieid int, title text, avg float, PRIMARY KEY(year, month, day, movieid));

3. Datos de tweets



En la siguiente ilustración podemos observar el flujo que siguen los datos de tweets dentro de la aplicación; a continuación lo detallaremos:

1. Datos, existen dos posibles entradas:
    1. Dataset, como necesitamos histórico he ido recopilando tweets en formato JSON durante el último mes de los tweets que hablan sobre las películas que hay actualmente en cartelera.
    2. A futuro sería quitar el productor de Kafka y utilizar directamente el conector que tiene Spark con tweeter para obtener en real time los tweets.
2. Capa de recolección de datos, en Kafka se ha realizado:
    1. Creación del topic tweet: bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic tweet --partitions 1 --replication-factor 1
    2. Se ha creado el productor ProducerTweet; este obtiene la información en formato JSON y la serializa en la clase tweet. Ha sido necesario crear el serializador y deserializador correspondiente.
3. Capa de procesamiento de datos, en Spark Streaming se ha realizado:
    1. Para que Spark Streaming pueda consumir de Kafka se ha metido la dependencia al conector spark-streaming-kafka-0.8_2.11
    2. Para que Spark pueda almacenar la información en Cassandra se ha metido la dependencia al conector spark-cassandra-connector
    3. Se ha creado el objeto StreamingTweetJob que será el encargado de:
        1. Consumir el topic tweet de kafka
        2. Guardar la información en bruto en HDFS en el directorio hdfs://localhost:9000/pfm/data/tweet/
        3. Obtenemos el sentimiento de cada tweet y generamos un nuevo RDD de objetos tweet que se almacena en la tabla pfm.tweet de Cassandra.
        Pruebas que se hicieron para calcular el Sentiment Analysis:
            1. Inicialmente se hicieron pruebas con la librería edu.standford.nlp pero el annotators sentiment no esta disponible para Español.
            2. El algoritmo de clasificación Naive Bayes de Spark ML lo ponían muy bien para esta situación pero al final se decidió no utilizar porque ya habíamos utilizado el de recomendación.
            3. Finalmente, aunque no es la mejor solución se ha utilizado una lista de palabras indicadoras de opinión en español dependientes del dominio; en este caso de críticas de cine. La lista está formada por 2.535 palabras positivas y 5.639 palabras

negativas. http://timm.ujaen.es/recursos/esol/
Se ha definido el método getTweetSentiment que se encarga de obtener el número de palabras positivas y negativas que aparecen en el texto de tweet y dependiendo de estos valores asignar los valores: positive, negative o neutral.

4. Guardamos este nuevo RDD en HDFS en el directorio hdfs://localhost:9000/pfm/etl/tweet/tweetsentiment_ en formato Parquet para que sea consumido posteriormente por el proceso Batch. Parquet es un formato de almacenamiento en columnas de código abierto para Hadoop. Soporta esquemas de codificación y compresión eficientes.
5. Procedemos al calculo del KPI Sentiment Analysis por hastag . En este punto como los cálculos se hacen por cada Dstream es necesario validar antes de guardar el cálculo en la view si ese row existe previamente para actualizar en lugar de insertar. Se define como clave (hastag, sentiment) se agrupa por dicha clave y se hace un conteo. Los resultados se almacenan en la view "stream_view_hastag_by_sentimental".
6. Capa de almacenamiento, como el keyspace ya se había creado anteriormente únicamente se han creado las views:
    1. CREATE TABLE tweet(id text PRIMARY KEY, text text, user text, timestamp timestamp, lang text, hastag text, sentiment text);
    2. CREATE TABLE stream_view_hastag_by_sentimental(hastag text, sentimental text, total int, PRIMARY KEY(hastag, sentimental));

4. Capa de procesamiento de datos, en Spark (batch layer) se ha realizado:
    1. Utilizando la API de Spark SQL obtenemos la información del directorio HDFS hdfs://localhost:9000/pfm/etl/tweet/tweetsentiment_
    2. Procedemos al calculo del KPI; como los cálculos se hacen sobre todos los datos aquí no se tiene en cuenta si existe o no. Los KPIs que hemos calculado son los siguientes:
        1. Sentiment Analysis por hastag: aplicamos la consulta SELECT hastag, sentiment, count(*) as total FROM tweet GROUP BY hastag, sentiment. Los resultados se almacenan en la view "batch_view_hastag_by_sentimental".
    3. Capa de almacenamiento, como el keyspace ya se había creado anteriormente únicamente se han creado las views:
        1. CREATE TABLE batch_view_hastag_by_sentimental(hastag text, sentimental text, total int, PRIMARY KEY(hastag, sentimental));