Lab 6 – Kafka en Twitter

CC5212-1 – April 23, 2025

Detectaremos terremotos usando Twitter y Kafka. Para esto, trabajaremos con datos de Twitter del 19 de septiembre de 2017; específicamente, tenemos una muestra del 1% de los datos de Twitter, con un total de 4,905,393 (re)tweets.[1]

En el servidor, el archivo `/data/uhadoop/shared/twitter/tweets_20170919.tsv.gz` contiene información sobre tweets, con las siguientes columnas: (1) datetime retrieved, (2) datetime written, (3) id of tweet, (4) user id, (5) tweet type, (6) language detected, (7) tweet text, (8) times retweeted. Algunos tweets son retweets: en esos casos, las columnas (2) y (3) hacen referencia al tweet original, por lo que las fechas en (2) pueden ser anteriores al día seleccionado. Para ver una muestra, usa el command: `zcat /data/uhadoop/shared/twitter/tweets_20170919.tsv.gz | more`.

Desde u-cursos, descarga el proyecto Java `mdp-lab06.zip`. En este archivo encontrarás código de ejemplo para comenzar a trabajar con Kafka. Se proporciona un script `build.xml` para ayudarte a compilar un JAR (de manera similar a los laboratorios anteriores).

- En `KafkaExample`, proporcionamos un ejemplo básico de Kafka. Usamos un thread para ejecutar un producer que envía un record a Kafka cada segundo, y luego ejecutamos un consumer para recuperar e imprimir los records desde Kafka. Intentémoslo! Compila el proyecto y copia el archivo jar a tu directorio en el sistema de archivos local del master server del clúster (`cluster-01`) (igual que hiciste anteriormente con Hadoop). En la conexión SSH, dentro de `cluster-01`, accede a tu directorio que contiene el archivo jar y ejecuta el comando: `java -jar mdp-kafka.jar KafkaExample`. Ah, Pero el sistema solicita un topic, el cual debemos crear primero:

  – `kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic` GROUPNAME`-example`

  Donde GROUPNAME es cualquier nombre único para tu grupo. Nota: Aquí podemos configurar el replication factor (factor de replicación) y el número de partitions (particiones). En el laboratorio, siempre trabajaremos con 1 para mantener las cosas más simples. Sin embargo, si configuras valores más altos, podrás utilizar más máquinas. Por cierto, si quieres ver los topics activos, usa el siguiente comando:

  – `kafka-topics.sh --list --zookeeper localhost:2181`

  Ahora ejecuta el ejemplo nuevamente con un topic: `java -jar mdp-kafka.jar KafkaExample` GROUPNAME`-example`. Se imprimirá cada segundo un mensaje con el siguiente formato:

  GROUPNAME`-example [0] offset=`n`, timestamp="`*timestamp*`", key=`n`, value="`*date*`"`

  Cuando te aburras, finaliza el proceso con Ctrl +C y elimina el topic:

  – `kafka-topics.sh --delete --zookeeper localhost:2181 --topic` GROUPNAME`-example`

- A continuación, ejecutaremos un Kafka producer para simular un flujo de tweets desde nuestro archivo. El código ya está implementado. Primero, crea tu propio topic para los tweets llamado GROUPNAME`-tweets` (como antes, puedes configurar replication y partitions en 1). Ahora ejecuta (todo como un solo comando – y revisa el código fuente mientras esperas):

  – `java -jar mdp-kafka.jar TwitterSimulator /data/uhadoop/shared/twitter/tweets_20170919.tsv.gz` GROUPNAME`-tweets 1000`

  El proceso tomará uno o dos minutos en completarse, aunque no parecerá ocurrir nada interesante. Pero lo que realmente está sucediendo es que el código tiene un producer enviando registros de tweets a tu topic de Kafka. Lamentablemente, aún no hay ningún consumer suscrito a ese topic. Observa el último argumento: `1000`. Este es el factor de aceleración. Como queremos procesar los tweets de un día completo pero no esperar 24 horas, el valor 1000 indica que aceleraremos el tiempo por un factor de 1000 (un segundo se convierte en un milisegundo).

[1] Agradecemos a Hernán Sarmiento por recolectar y proporcionar los datos!

¿Cuánto tiempo tomará ejecutar un día? También ten en cuenta que tenemos una muestra del 1% de Twitter, por lo que si seleccionamos `100`, estaremos simulando el rendimiento en tiempo real de Twitter. Seleccionar `1000` significa que estamos 10 veces más rápidos que el flujo real de Twitter para ese día en particular.

- Ahora vamos a utilizar esos tweets para intentar detectar terremotos. Echa un vistazo al código en `PrintEarthquakeTweets`. El código incluye un consumer que lee mensajes de un topic (pasado como argumento) e imprime en consola todos los records que contengan sub-string relacionadas con terremotos. Para ejecutarlo, usa el siguiente comando:

  - `java -jar mdp-kafka.jar PrintEarthquakeTweets` GROUPNAME`-tweets`

  Decepción! No está ocurriendo nada. El problema es que, por defecto, un consumer lee desde el punto actual del stream y el flujo de tweets ha terminado. Intenta dejar `PrintEarthquakeTweets` ejecutándose, abre otro terminal en `cluster-01` y ejecuta `TwitterSimulator` nuevamente mientras monitoreas la salida de `PrintEarthquakeTweets`.

  Algunos tweets de salida no parecen estar relacionados con terremotos, y otros hablan de terremotos antiguos. Si queremos detectar terremotos en Twitter, necesitaremos identificar un *burst* repentino (aumento significativo) de tweets sobre terremotos en un mismo momento. Entonces, tu tarea final es crear tu propio Kafka producer y consumer(s) para hacer esto.

  Filtro de terremotos: Crea una clase llamada `EarthquakeFilter` basada en `PrintEarthquakeTweets` que reciba un input topic como antes, pero en lugar de imprimir en la salida estándar, añade otro argumento que acepte el nombre de un output topic y luego cree un producer para escribir en ese topic. En el nuevo output topic (partición 0), la clase debe escribir tweets relevantes para terremotos (según los criterios de `PrintEarthquakeTweets`), copiando el timestamp, key y value del record de entrada al record de salida. Puedes basarte en los ejemplos de `KafkaExample` and `TwitterSimulator`.Cuando estés listo, construye el jar, cópialo al servidor, crea un nuevo topic para los tweets sobre terremotos en tu grupo, y ejecuta el código. (De nuevo, parecerá aburrido porque aún no hay nada que consuma los tweets...).

  Detector de Burst (ráfagas): Crea una clase con método main llamada `BurstDetector` que utilice un consumer para leer de un input topic y detecte ráfagas (bursts) de records. Definimos un (x, y)-*burst* como la recepción de x tweets en máximo y segundos. **Para medir el tiempo y, debes leer el timestamp del record, el cual viene expresado en milisegundos.**; Esto indica la hora real en la que se escribió el tweet. Luego, definimos un *event* de la siguiente manera. Al inicio del stream asumimos que no estamos dentro de un evento. Si detectamos un (x, y)-burst de mensajes y no estamos actualmente en un evento, se declara el inicio de un nuevo evento (pasando a estado "evento activo"). Si la frecuencia disminuye a $(x, 2y)$ o menos (es decir, se reciben x tweets en al menos $2y$ segundos), mientras hay un evento activo, declaramos que el evento actual finaliza (pasando a estado "sin evento").[2] Por ejemplo, considera el flujo mostrado en la tabla a continuación con el offset n y timestamp t (no mostramos la key, value, etc., por brevedad). Supongamos que estamos buscando eventos para $(x, y) = (4, 4)$. El primer evento comienza en $n = 2$, y puede ser detectado cuando recibimos $n = 5$. Esto se debe a que $t_5 - t_2 = 8 - 4 \leq 4$; es decir, hemos visto $x = 4$ mensajes en (como máximo) $y = 4$ segundos, y no estábamos previamente en un evento. El primer evento termina en $k = 10$ porque $t_{10} - t_7 = 20 - 10 \geq 8$; es decir, aún estábamos en el primer evento, pero ahora hemos recibido $x = 4$ tweets en al menos $2y = 8$ segundos. El segundo evento comienza en $k = 11$ y se detecta en $k = 14$. El flujo termina con el segundo evento aún activo.[3]

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
t	1	2	4	6	7	8	9	10	12	16	20	25	27	27	28	30	33

  Para implementar el consumer que detecta eventos en el output topic de `EarthquakeFilter` para $(x, y) = (50, 50)$. Imprime en la consola un mensaje al inicio del evento, incluyendo el mensaje y el timestamp (opcionalmente formateado como una fecha legible para humanos) del inicio de cada evento, un identificador numérico para el evento (1 para el primer evento, 2 para el segundo, etc.), y opcionalmente la tasa de burst (los valores de x e y al detectar el inicio del evento, lo cual puede ayudarte a depurar el código). También deberías imprimir en la consola un mensaje para el final de cada evento, con el identificador del evento

[2] Separar la tasa de inicio/fin de esta manera evita que se activen múltiples eventos debido a la oscilación de la tasa en ambos lados del umbral cercano a (x, y).

[3] Nota: Bajo esta definición, los eventos pueden superponerse. Por ejemplo, si modificamos ambos valores t_{11} and t_{12} a 20, se detectaría un segundo evento comenzando en t_9 y t_9 pertenecería tanto al primer como al segundo evento.

que termina, y opcionalmente la tasa de burst (los valores de x e y al detectar el fin del evento). El primer mensaje para el primer evento debe ser "`JUST FELT MY FIRST EARTHQUAKE AND I DID NOT CARE FOR IT`" alrededor de las 06:22 (las horas están en UTC). ¿También detectaste el terremoto de Puebla de 2017? Busca un evento con un mensaje en español alrededor de la hora en que ocurrió.

Hint: Piensa en una cola FIFO (hint: `LinkedList` en Java)que almacena x mensajes (o como máximo x mensajes al principio). Cuando la cola FIFO se haya llenado (es decir, hayas leído al menos x mensajes) y la diferencia de tiempo entre el elemento más antiguo y el más nuevo sea menor o igual a y, ingresa a un evento (si aún no estás en uno) e imprime el mensaje más antiguo. Cuando la misma diferencia sea mayor o igual a $2y$, termina el evento (si ya estás en uno). Recuerda que los timestamps estarán en milisegundos, por lo que puedes usar $1000y$.

Entrega: Debes subir a u-cursos tu solución de `EarthquakeFilter.java` y `BurstDetector.java`. También envía un tercer archivo `results.txt` con el registro de la consola con los eventos detectados por `BurstDetector.java`.