Lab 6 – Kafka vs. Twitter

CC5212-1 – April 13, 2022

We will detect earthquakes with Twitter and Kafka. For this, we will work with Twitter data from September 19th, 2017; specifically we have a 1% sample of Twitter data with a total of 4,905,393 (re)tweets.[1]

On the server, the file `/data/uhadoop/shared/twitter/tweets_20170919.tsv.gz` contains information about tweets, where the columns are as follows: (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. Some of the tweets are retweets, where (2) and (3) refer to the original tweet; hence you may see dates in (2) that are older than the selected date. To peek, use `zcat /data/uhadoop/shared/twitter/tweets_20170919.tsv.gz | more`.

From u-cursos, download the project `mdp-lab06.zip`. Here you will find some example code to get you started with Kafka. A `build.xml` script is provided to help you build a jar (as in the previous labs).

- Have a look at `KafkaExample`, which offers a Hello-World-style example for Kafka. Let's give it a try! Build the project and copy the jar over to the server. Run `java -jar mdp-kafka.jar KafkaExample`. Ah, but it asks for an argument: a *topic* on Kafka. What should we put? Let's create a new topic; note that the following is all one command.

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

  where GROUPNAME is any unique name for your group. Note that here we can set the replication factor and number of partitions. In the lab we will always work with 1 to keep things a bit simpler, but if you set these higher, you can use more machines. By the way, if you want to see the active topics:

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

  Now we are ready to run our example again, this time with a topic:

  – `java -jar mdp-kafka.jar KafkaExample` GROUPNAME`-example`

  Not the most exciting example ever, but hey, it's a start! Okay, let's delete that topic we don't need any more:

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

- Okay, next we will run a Kafka producer to simulate a stream of tweets from our file. The code is already done. First, create your own topic for tweets called GROUPNAME`-tweets` as before (you can set replication and partitions to 1 again). Now run (and have a look at the source code while you wait):

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

  It will take a minute or two to finish though not much of interest seems to happen. But what is happening is that the code is writing Tweets to your Kafka topic. Unfortunately nothing is subscribed to that topic yet. Note the last argument: `1000`. This is the speed-up. We want to process the tweets of an entire day but don't want to wait an entire day, so 1000 specifies to speed up time by a factor of 1000 (one second becomes one millisecond). How long will it take to run a day? Also note that we have a 1% sample of Twitter, so if we select `100`, we are simulating real-time Twitter throughput. Selecting `1000` then means we are 10 times faster than Twitter's real stream.

[1] Thanks to Hernán Sarmiento for collecting/providing the data!

- So let's do something with those tweets; something earthquake related. Have a look at the code in `PrintEarthquakeTweets`. It reads from a topic (passed as an argument) and prints out any records that have an earthquake-related sub-string in them. Let's try it out:
  - `java -jar mdp-kafka.jar PrintEarthquakeTweets GROUPNAME-tweets`

  Anti-climax! Nothing is happening. The issue is that by default a consumer will read from the current point of the stream and the tweet stream is finished. Try running `TwitterSimulator` again while `PrintEarthquakeTweets` is waiting.

  Some output tweets don't seem so related to earthquakes, and some are about old earthquakes. If we want to detect earthquakes on Twitter, we'll need to detect a sudden *burst* of earthquake tweets at the same time. So your final task is to code your own Kafka producer and consumer(s) to do this!

  Consumer/Producer: Create a class called `EarthquakeFilter` based on `PrintEarthquakeTweets`, but instead of printing to standard out, add another argument that accepts the name of an output topic, and then creates a producer to write to that topic. You can follow the examples in `KafkaExample` and `TwitterSimulator`. When ready, build the code, create a new topic for earthquake tweets in your group, and run the code. (Again it's a bit boring because nothing is there to consume the tweets yet ...)

  Consumer: Create a main method class `BurstDetector`, which uses a consumer to read from an input topic and then detects a burst of records. We define a *burst* (x, y) as receiving x tweets in at most y seconds. We then define an *event* as follows. At the start of the stream we assume that we are not in an event. If we detect an (x, y) burst of messages, and we are not currently in an event, we say that a new event starts (we are now in an event). If the rate slows to $(x, 2y)$ or slower (x tweets take at least $2y$ seconds), and we are in an event, we say that the current event ends (we are no longer in an event).[2] For example, consider the stream shown in the table below with offset n and timestamp t (we do not show the key, value, etc., for brevity). Assume we are looking for events for $(x, y) = (4, 4)$. The first event starts at $n = 2$, and can be detected when we receive $n = 5$. This is because $t_5 - t_2 = 8 - 4 \leq 4$; i.e., we have seen $x = 4$ messages in (at most) $y = 4$ seconds, and we were not previously in an event. The first event ends at $k = 10$ because $t_{10} - t_7 = 20 - 10 \geq 8$; i.e., we were still in the first event, but now we have received $x = 4$ tweets in at least $2y = 8$ seconds. The second event starts at $k = 11$ and is detected at $k = 14$. The stream ends with the second event still active.

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	26	27	28	30	33

  You should implement your consumer to detect events over the output topic of `EarthquakeFilter` for $(x, y) = (50, 50)$, printing out the message for the start of each event.

  Hint: Think about a FIFO queue (hint: `LinkedList` in Java) storing x messages (or at most x messages at the start), that can compare the difference in time between the oldest and newest elements against y (note that the timestamps will be in milliseconds, so you can use $1000y$).

- SUBMIT to u-cursos your `EarthquakeFilter.java` and `BurstDetector.java` solution. Also submit as `results.txt` the number of minor/major bursts detected, followed by the messages printed for the start of each major/minor burst detected (e.g., in order of timestamp).

[2] Separating the start/end rate in this way avoids multiple events being triggered by the rate oscillating on both sides of the threshold close to (x, y).