DataEng: Data Transport Activity

[this lab activity references tutorials at confluence.com]

Make a copy of this document and use it to record your results. Store a PDF copy of the document in your git repository along with your code before submitting for this week. For your code, you create several producer/consumer programs or you might make various features within one program. There is no one single correct way to do it. Regardless, store your code in your repository.

The goal for this week is to gain experience and knowledge of using a streaming data transport system (Kafka). Complete as many of the following exercises as you can. Proceed at a pace that allows you to learn and understand the use of Kafka with python.

A. Initialization

1. Get your cloud.google.com account up and running
    a. Redeem your GCP coupon
    b. Login to your GCP console
    c. Create a new, separate VM instance
2. Follow the Kafka tutorial from project assignment #1
    a. Create a separate topic for this in-class activity
    b. Make it “small” as you will not want to use many resources for this activity
    c. Get a basic producer and consumer working with a Kafka topic as described in the tutorials.
3. Create a sample breadcrumb data file (named bcsample.json) consisting of a sample of 1000 breadcrumb records. These can be any records because we will not be concerned with the actual contents of the breadcrumb records during this assignment.
4. Update your producer to parse your sample.json file and send its contents, one record at a time, to the kafka topic.
5. Use your consumer.py program (from the tutorial) to consume your records.

B. Kafka Monitoring

1. Find the Kafka monitoring console for your topic. Briefly describe its contents. Do the measured values seem reasonable to you? **Yes**
2. Use this monitoring feature as you do each of the following exercises.

C. Kafka Storage

1. Run the linux command “wc bcsample.json”. Record the output here so that we can verify that your sample data file is of reasonable size.
   5887
2. What happens if you run your consumer multiple times while only running the producer once?
   After consuming the initial producer output, the consumer repeatedly prints the message, "Waiting for message ...".
3. Before the consumer runs, where might the data go, where might it be stored?
   Not exactly sure -- in the cloud, haha. I guess it lives somewhere in the Kafka cluster instance created in Confluent?
4. Is there a way to determine how much data Kafka/Confluent is storing for your topic? Do the Confluent monitoring tools help with this?
   Yes, there is a Storage chart in the Cluster overview.
5. Create a “topic_clean.py” consumer that reads and discards all records for a given topic. This type of program can be very useful during debugging.
   This looks just like consumer.py, but in *if msg is None:* I replaced the *continue* with *break* so after consuming all messages the consumer runs *consumer.close().*

D. Multiple Producers

1. Clear all data from the topic
   OK.
2. Run two versions of your producer concurrently, have each of them send all 1000 of your sample records. When finished, run your consumer once. Describe the results.
   The producer runs so quickly that it's hard to run 2 instances concurrently (in the sense that they're both producing at the same time in an interleaved fashion), but I ran one right after another and the consumer appears to have consumed all messages for the first producer, then all messages from the second producer.

E. Multiple Concurrent Producers and Consumers

1. Clear all data from the topic
   OK. I didn't run topic_clean.py, but the topic seems to be cleaned once the there are no consumers running (?).
2. Update your Producer code to include a 250 msec sleep after each send of a message to the topic.

OK.

3. Run two or three concurrent producers and two concurrent consumers all at the same time.
   Now we get interleaved producers!
4. Describe the results.
   Both producers are producing, but it looks like only the first consumer is consuming. It's taking whatever messages enter the queue, not all from one producer first, then the next producer. The second consumer is waiting patiently.

F. Varying Keys

1. Clear all data from the topic
   OK. I didn't even stop the producers this time and, looking at Confluent > Topics, the topic appears to have been emptied.

So far you have kept the “key” value constant for each record sent on a topic. But keys can be very useful to choose specific records from a stream.

2. Update your producer code to choose a random number between 1 and 5 for each record’s key.
   OK.
3. Modify your consumer to consume only records with a specific key (or subset of keys).
   I had to change the randomly generated number used as each record's key to a str, as that's what produce() requires. Then I changed the first conditional to *if msg is None or msg.key() is not "1":* , but the consumer doesn't seem to be consuming ANY messages (based on the print statements). After producing 1000 messages, I went to cloud.confluent to look at Consumers and it said the Consumer was 0 messages behind (I checked when consumer.py was still running and after I stopped it). I would expect the messages to still be there for other Consumers to consume, so am very confused. ...
   OK, I see why the messages were being consumed -- the *if* statement comes AFTER *consumer.poll()*. But I still don't understand why the Consumer didn't seem to enter the *else*, even when the msg.key() was *"1"*.
   OK, I see that msg.key() returns the key prepended with b (for bytes or something like that, presumably). So, I got that part working. But I think I need to change the logic so the message doesn't get consumed if the key is not correct -- so that the message can be consumed by the correct

Consumer later on. On second thought, I seem to remember that messages are left for others to consume later, so I'll hold off for now. ...
OK, I reread the instructions and indeed we only want to <u>consume</u> records with a certain key. Changing the logic. ... So we need to "precheck" the key without consuming it.

4. Attempt to consume records with a key that does not exist. E.g., consume records with key value of “100”. Describe the results
   Not sure how to do this (?).

5. Can you create a consumer that only consumes specific keys? If you run this consumer multiple times with varying keys then does it allow you to consume messages out of order while maintaining order within each key?
   Haven't found a way to do this and based on a thread in Slack, others have struggled to find a way. So, I've thrown in the towel on this -- it seems like it's not possible.

G. Producer Flush

The provided tutorial producer program calls “producer.flush()” at the very end, and presumably your new producer also calls producer.flush().

1. What does Producer.flush() do?
2. What happens if you do not call producer.flush()?
3. What happens if you call producer.flush() after sending each record?
4. What happens if you wait for 2 seconds after every 5th record send, and you call flush only after every 15 record sends, and you have a consumer running concurrently? Specifically, does the consumer receive each message immediately? only after a flush? Something else?

H. Consumer Groups

1. Create two consumer groups with one consumer program instance in each group.
2. Run the producer and have it produce all 1000 messages from your sample file.
3. Run each of the consumers and verify that each consumer consumes all of the 50 messages.
4. Create a second consumer within one of the groups so that you now have three consumers total.
5. Rerun the producer and consumers. Verify that each consumer group consumes the full set of messages but that each consumer within a consumer group only consumes a portion of the messages sent to the topic.

I. Kafka Transactions

6. Create a new producer, similar to the previous producer, that uses transactions.
7. The producer should begin a transaction, send 4 records in the transactions, then wait for 2 seconds, then choose True/False randomly with equal probability. If True then finish the transaction successfully with a commit. If False is picked then cancel the transaction.
8. Create a new transaction-aware consumer. The consumer should consume the data. It should also use the Confluent/Kaka transaction API with a “read_committed” isolation level. (I can’t find evidence of other isolation levels).
9. Transaction across multiple topics. Create a second topic and modify your producer to send two records to the first topic and two records to the second topic before randomly committing or canceling the transaction. Modify the consumer to consume from the two queues. Verify that it only consumes committed data and not uncommitted or canceled data.