Neha Agrawal

FYI – my responses are inlined below in italic.

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. By “small” I mean that you should choose medium or minimal options when asked for any configuration decisions about the topic, cluster, partitions, storage, anything. GCP/Confluent will ask you to choose the configs, and because you are using a free account you should opt for limited resources where possible.
    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?

   We can find kafka monitoring console by logging in to confluent cloud →Our Cluster→Topics→topic-name→ Metrics
   We see three tabs there-
   - *Production which shows bytes produced per sec in a selected time range.*
   - *Consumption which shows bytes consumed per sec in a selected time frame.*
   - *Consumer Lag – which shows number of messages behind.*

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.

2. What happens if you run your consumer multiple times while only running the producer once?

   This means no message is available for consumption.

3. Before the consumer runs, where might the data go, where might it be stored?

   *The Kafka cluster stores stream of published record in categories called **topic** until they have been consumed by the consume using a configurable retention period.*

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, we can go to the confluent cloud web UI and navigate to data flow and inspect our topic. On the right side, we can see total throughput, bytes in and out per partition and bytes written per producer.

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.

D. Multiple Producers

1. Clear all data from the topic
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 consumer consumes the records from both the producer instances. It consumes total 2000 records, 1000 from each producer.

E. Multiple Concurrent Producers and Consumers

1. Clear all data from the topic
2. Update your Producer code to include a 250 msec sleep after each send of a message to the topic.
3. Run two or three concurrent producers and two concurrent consumers all at the same time.
4. Describe the results.

I created a topic with single partition.
Observation: The results published by all the producers were consumed by a single consumer

Conclusion: At any time, a partition can only be consumed by a single consumer. Two consumers can’t consume records from a single partition at the same time.

F. Varying Keys

1. Clear all data from the topic

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.
3. Modify your consumer to consume only records with a specific key (or subset of keys).
4. Attempt to consume records with a key that does not exist. E.g., consume records with key value of "100". Describe the results

My consumer code looks like the one shown below to allow records with specific key to be consume and discard the other records:

```
# Process messages
total_count = 0
try:
    while True:
        msg = consumer.poll(1.0)
        if msg is None:
            # No message available within timeout.
            # Initial message consumption may take up to
            # `session.timeout.ms` for the consumer group to
            # rebalance and start consuming
            print("Waiting for message or event/error in poll()")
            continue
        elif msg.error():
            print('error: {}'.format(msg.error()))
        else:
            # Check for Kafka message
            record_key = msg.key().decode('utf-8')
            if(record_key == '100'):
                record_value = msg.value().decode('utf-8')
                data = json.loads(record_value)

                total_count += 1
                print("Consumed record with key {}, value {}, \
                  and updated total count to {}"
                  .format(record_key,  record_value, total_count))
            else:
                print("Waiting for record with specific key")
```

So in this case, if consumer tries to consume record with record_key=100, it will directly go to the else clause and print "Waiting for record with specific key".

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?

Yes, we can create a consumer that only consumes specific keys.

I created a topic with a single partition – Once a consumer is subscribed to a partition in a topic, it consumes all the messages. Here we are just handling the messages with a specific key and discarding the others using if – else statement.

So, if we run this consumer second time with different key/s, it will wait for new messages to be produced by the producer.

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?

   Wait for all messages in the Producer queue to be delivered. This is a convenience method that calls poll() until len() is zero or the optional timeout elapses.

2. What happens if you do not call producer.flush()?

   produce() is asynchronous, it simply enqueues our message on an internal queue for later transmission to the broker.

   When we call poll(0) directly after produce() it is highly unlikely that the message we just produced has been sent to the broker, processed, and an ack response received from the broker in that short time. So we will most likely only see delivery reports from previous messages at that point.

   Adding flush() before exiting will make the client wait for any outstanding messages to be delivered to the broker.

   Source: https://github.com/confluentinc/confluent-kafka-python/issues/137

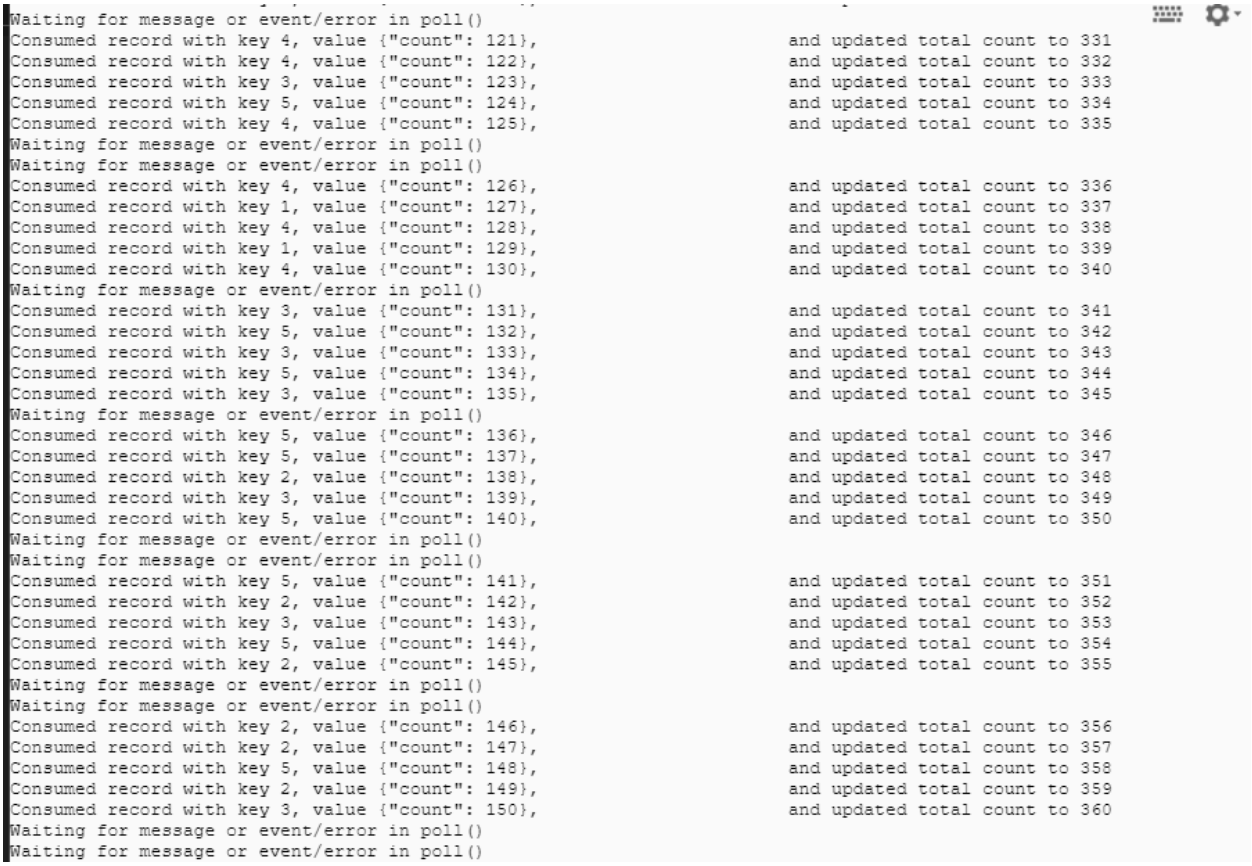
3. What happens if you call producer.flush() after sending each record?

   If we add flush() after each produce() call you are effectively implementing a sync producer

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?

We saw the records are consumed in chunks of 5. This is how the consumer window looks like:



```
Waiting for message or event/error in poll()
Consumed record with key 4, value {"count": 121},                    and updated total count to 331
Consumed record with key 4, value {"count": 122},                    and updated total count to 332
Consumed record with key 3, value {"count": 123},                    and updated total count to 333
Consumed record with key 5, value {"count": 124},                    and updated total count to 334
Consumed record with key 4, value {"count": 125},                    and updated total count to 335
Waiting for message or event/error in poll()
Waiting for message or event/error in poll()
Consumed record with key 4, value {"count": 126},                    and updated total count to 336
Consumed record with key 1, value {"count": 127},                    and updated total count to 337
Consumed record with key 4, value {"count": 128},                    and updated total count to 338
Consumed record with key 1, value {"count": 129},                    and updated total count to 339
Consumed record with key 4, value {"count": 130},                    and updated total count to 340
Waiting for message or event/error in poll()
Consumed record with key 3, value {"count": 131},                    and updated total count to 341
Consumed record with key 5, value {"count": 132},                    and updated total count to 342
Consumed record with key 3, value {"count": 133},                    and updated total count to 343
Consumed record with key 5, value {"count": 134},                    and updated total count to 344
Consumed record with key 3, value {"count": 135},                    and updated total count to 345
Waiting for message or event/error in poll()
Consumed record with key 5, value {"count": 136},                    and updated total count to 346
Consumed record with key 5, value {"count": 137},                    and updated total count to 347
Consumed record with key 2, value {"count": 138},                    and updated total count to 348
Consumed record with key 3, value {"count": 139},                    and updated total count to 349
Consumed record with key 5, value {"count": 140},                    and updated total count to 350
Waiting for message or event/error in poll()
Waiting for message or event/error in poll()
Consumed record with key 5, value {"count": 141},                    and updated total count to 351
Consumed record with key 2, value {"count": 142},                    and updated total count to 352
Consumed record with key 3, value {"count": 143},                    and updated total count to 353
Consumed record with key 5, value {"count": 144},                    and updated total count to 354
Consumed record with key 2, value {"count": 145},                    and updated total count to 355
Waiting for message or event/error in poll()
Waiting for message or event/error in poll()
Consumed record with key 2, value {"count": 146},                    and updated total count to 356
Consumed record with key 2, value {"count": 147},                    and updated total count to 357
Consumed record with key 5, value {"count": 148},                    and updated total count to 358
Consumed record with key 2, value {"count": 149},                    and updated total count to 359
Consumed record with key 3, value {"count": 150},                    and updated total count to 360
Waiting for message or event/error in poll()
Waiting for message or event/error in poll()
```

The consumer receives each message after it is flushed.

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.