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

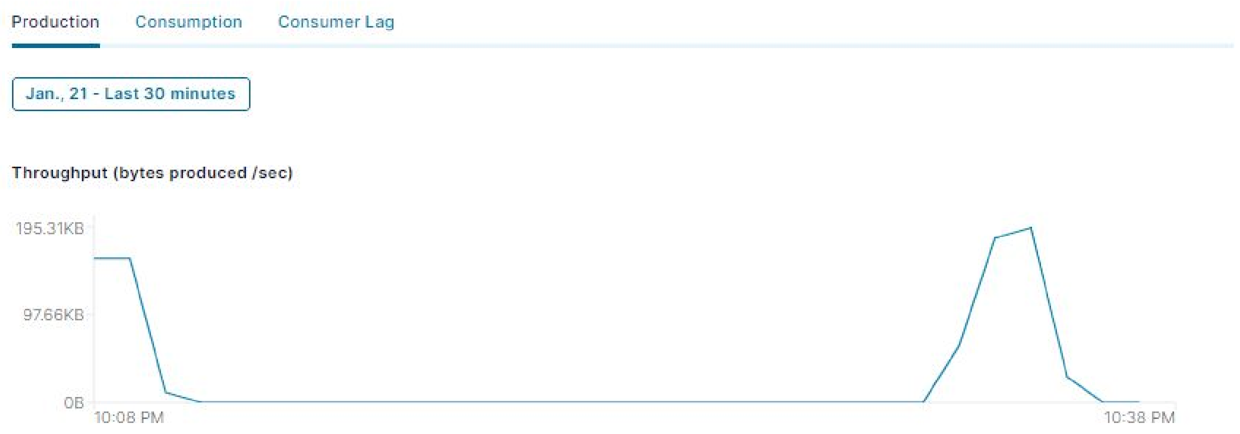
1. Get your cloud.google.com account up and running
    a. Redeem your GCP coupon
    b. Login to your GCP console
    c. Create a new, separate VM instance
2. Follow the Kafka tutorial from project assignment #1
    a. Create a separate topic for this in-class activity
    b. Make it “small” as you 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.

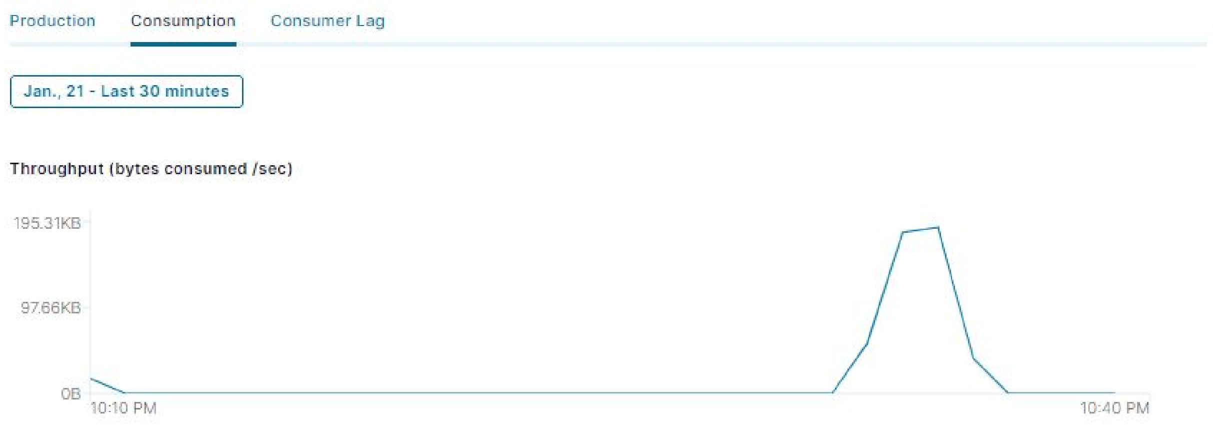
B. Kafka Monitoring

1. Find the Kafka monitoring console for your topic. Briefly describe its contents. Do the measured values seem reasonable to you?

   The topics section has the monitoring metrics. It has three metrics: Production, Consumption and consumer lag.

   The consumer throughput closely follows producer throughput.
   There is consumer lag ranging from 1100 to 0. It means that sometimes around 1000 messages wait in the queue.

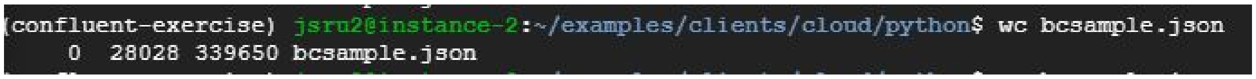




2. Use this monitoring feature as you do each of the following exercises.

C. Kafka Storage

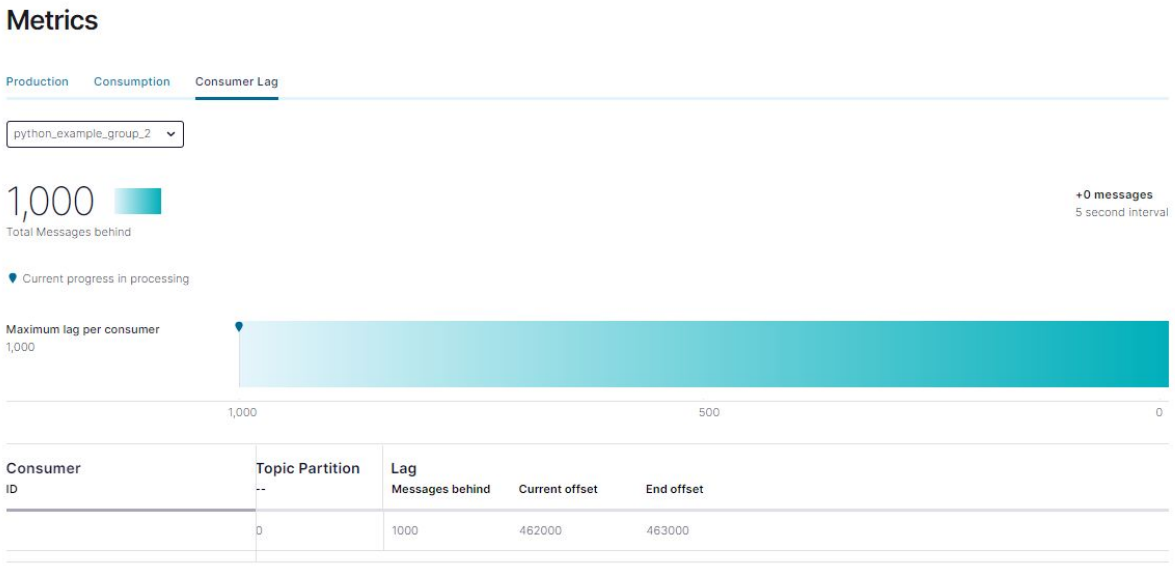
1. Run the linux command “wc bcsample.json”. Record the output here so that we can verify that your sample data file is of reasonable size.
   sni



```
(confluent-exercise) jsru2@instance-2:~/examples/clients/cloud/python$ wc bcsample.json
     0  28028 339650 bcsample.json
```

2. What happens if you run your consumer multiple times while only running the producer once?
   I didn’t understand the question properly hence trying to answer all possible scenarios.
   - Consumer multiple times. Sequentially: First consumer consumes all topics. The second time it is run, it waits for messages from the producer.
   - Same Consumer multiple times in parallel: First started consumer consumes all messages. The second instance of the consumer waits for messages, until the first one is stopped.
   - Two consumers with different group Ids run in parallel: Both consumers consume the messages parallely.

3. Before the consumer runs, where might the data go, where might it be stored?
   The data would go into the queue. The data is stored in the partition of the topic.



Metrics

Consumer ID	Topic Partition --	Lag Messages behind	Current offset	End offset
	0	1000	462000	463000

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?

No, Not much. I could not find a topic wise storage stats. Overall storage can be seen from cluster overview. The offset counts and throughput details in topic metrics gives us a fair idea of data each topic has.

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.

   I couldn’t find an API call to clear the messages. I went and changed the retention setting in the topic.

D. Multiple Producers

1. Clear all data from the topic
   I couldn’t find an API call to clear the messages. I went and changed the retention setting in the topic. I am sure this can be automated through CLI. Moving ahead keeping the time in mind.
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. (Reverted back retention settings)
   The message lag started with 2000 and dropped to zero quickly. Throughput of 2 producers and a single consumer almost remained the same.

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.

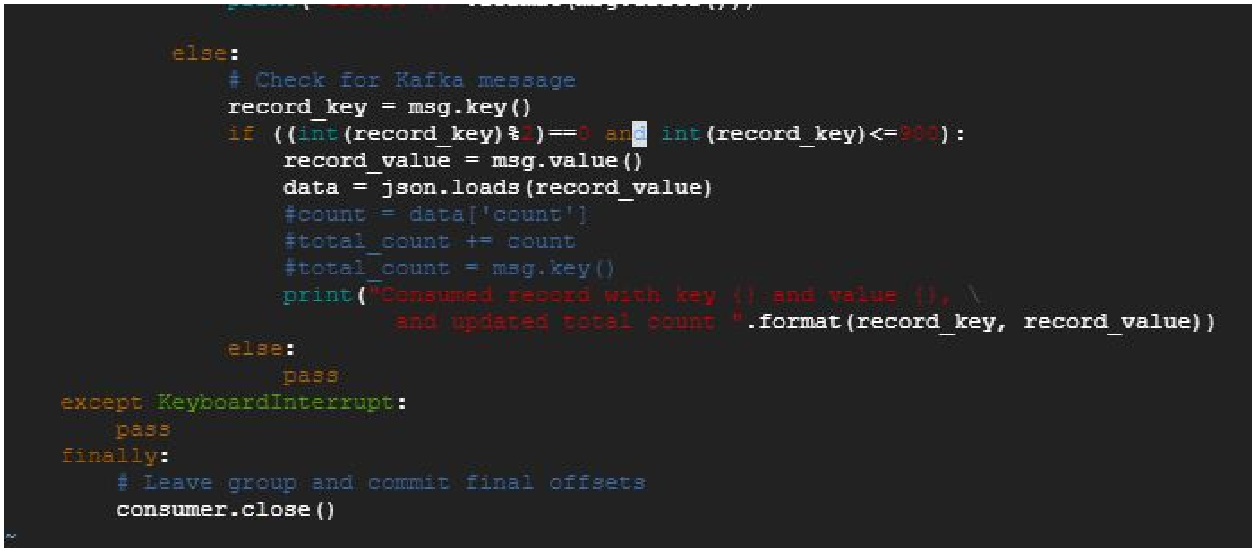
Only one consumer consumed the message. While the second instance of the consumer was just waiting for the message. Also the message lag upto 200 was observed when three producers were producing at constant rate and only one consumer was consuming it.

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
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?
   The following loops consumes messages whose keys are even number and are less than 9000. (my producer produces keys of range (1,1000). This consumes messages in the order of arrival skipping messages whose key values or odd.



```
            else:
                # Check for Kafka message
                record_key = msg.key()
                if ((int(record_key)%2)==0 and int(record_key)<=900):
                    record_value = msg.value()
                    data = json.loads(record_value)
                    #count = data['count']
                    #total_count += count
                    #total_count = msg.key()
                    print("Consumed record with key {} and value {}, \
                          and updated total count ".format(record_key, record_value))
                else:
                    pass
    except KeyboardInterrupt:
        pass
    finally:
        # Leave group and commit final offsets
        consumer.close()
```

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?
   Flush() call is used to make the producer synchronous. Flush blocks until the previously sent messages have been delivered (deliver all messages in queue and receive some sort of acknowledgement). What happens if you do not call producer.flush()?

If you do not call flush() you wouldn't know if messages in the producer queue have been delivered to the topic. The print statement after flush confirms that the job is done.

2. What happens if you call producer.flush() after sending each record?
   It would make the program wait sending another record until acknowledgement for the previous record is received.
3. 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?
   Consumers receive messages immediately because the messages could have been delivered during the wait time. Flush in this case doesn't have to wait at all since the messages must have been delivered and acknowledged.

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.