DataEng S24: PubSub

[this lab activity references tutorials at cloud.google.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 publisher/receiver 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 an asynchronous data transport system (Google PubSub). Complete as many of the following exercises as you can. Proceed at a pace that allows you to learn and understand the use of PubSub with python.

Submit: use the in-class activity submission form which is linked from the Materials page on the class website. Submit by 10pm PT this Friday.

A. [MUST] PubSub Tutorial

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. Complete this PubSub tutorial: link Note that the tutorial instructs you to destroy your PubSub topic, but you should not destroy your topic just yet. Destroy the topic after you finish the following parts of this in-class assignment.

B. [MUST] Create Sample Data

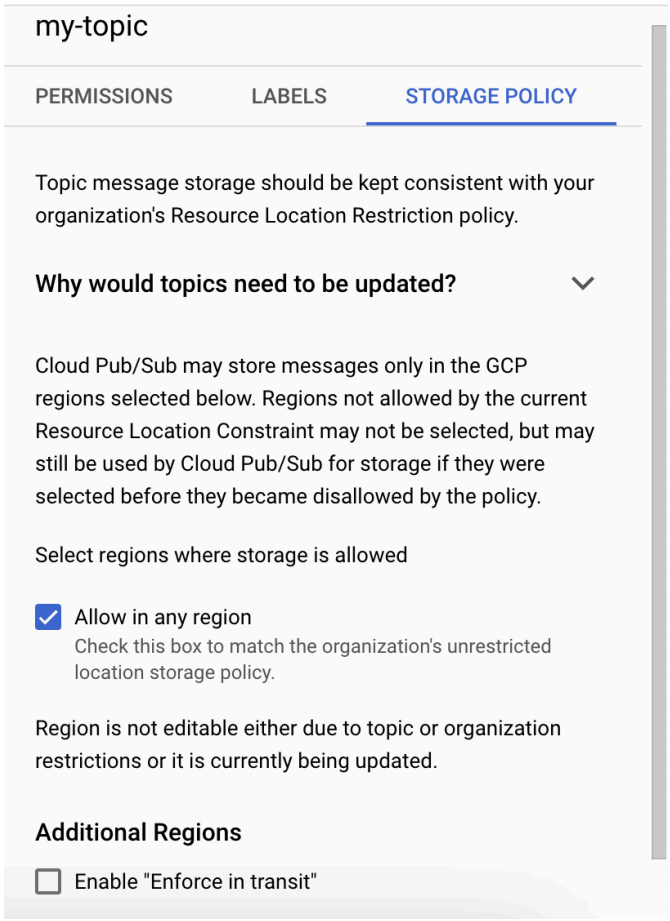
1. Get data from https://busdata.cs.pdx.edu/api/getBreadCrumbs for two Vehicle IDs from among those that have been assigned to you for the class project.
2. Save this data in a sample file (named bcsample.json)
3. Update the publisher python program that you created in the PubSub tutorial to read and parse your bcsample.json file and send its contents, one record at a time, to the my-topic PubSub topic that you created for the tutorial.
4. Use your receiver python program (from the tutorial) to consume your records.

C. [MUST] PubSub Monitoring

1. Review the PubSub Monitoring tutorial: link and work through the steps listed there. You might need to rerun your publisher and receiver programs multiple times to trigger enough activity to monitor your my-topic effectively.

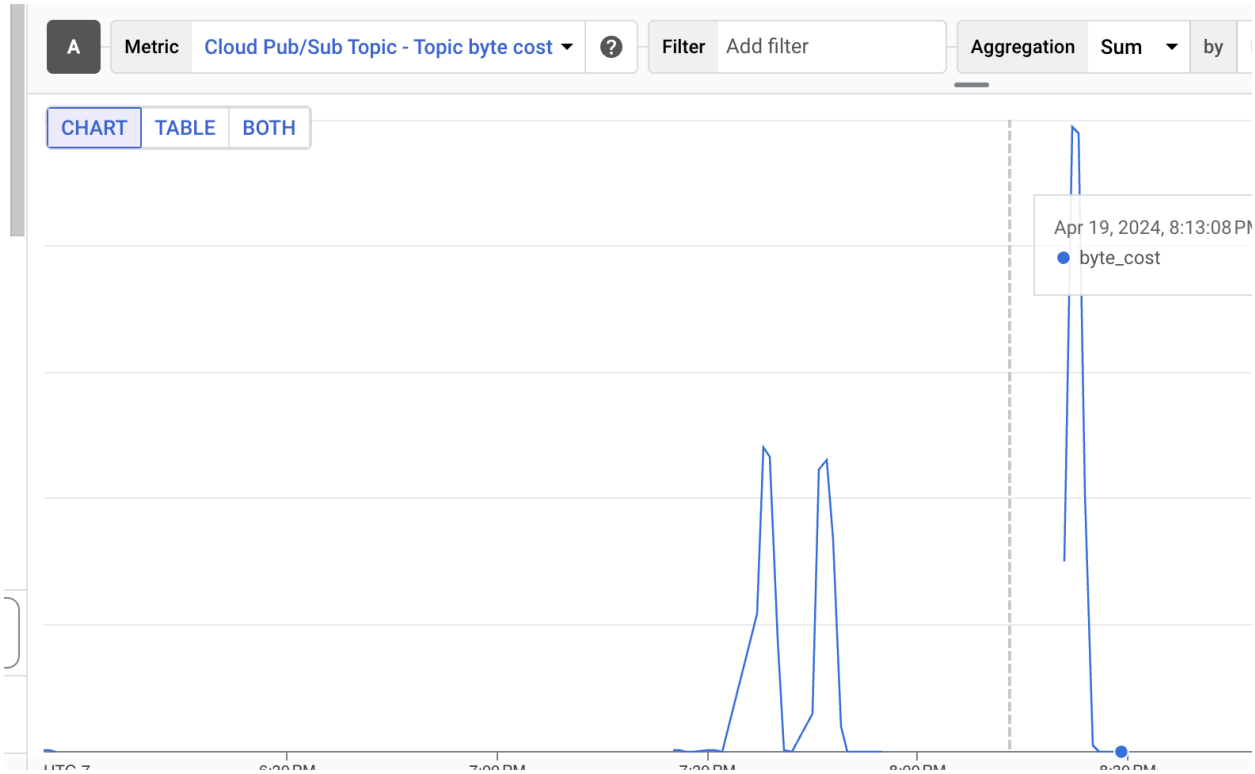
D. [MUST] PubSub Storage

1. What happens if you run your receiver multiple times while only running the publisher once?
   Receiver runs out of messages to acknowledge after a point. This effectively means that where there are 0 `unacknowledged_messages` in the Monitoring dashboard, running the receiver will be a NOOP.
2. Before the consumer runs, where might the data go, where might it be stored?
   Messages are stored in the Topic until a subscriber acknowledges and pulls them out. GCP Metric `byte_cost` helps to visualize how much data was utilized at any point of time. The topic (comprising the messages) is physically stored in any region since that's the default setting for the PSU Organization. However, this can be changed to any specific region if we'd like.



3. Is there a way to determine how much data PubSub is storing for your topic? Do the PubSub monitoring tools help with this?

Topic byte cost
Description
Cost of operations, measured in bytes. This is used to measure utilization for quotas.
Metric
pubsub.googleapis.com/topic/byte_cost

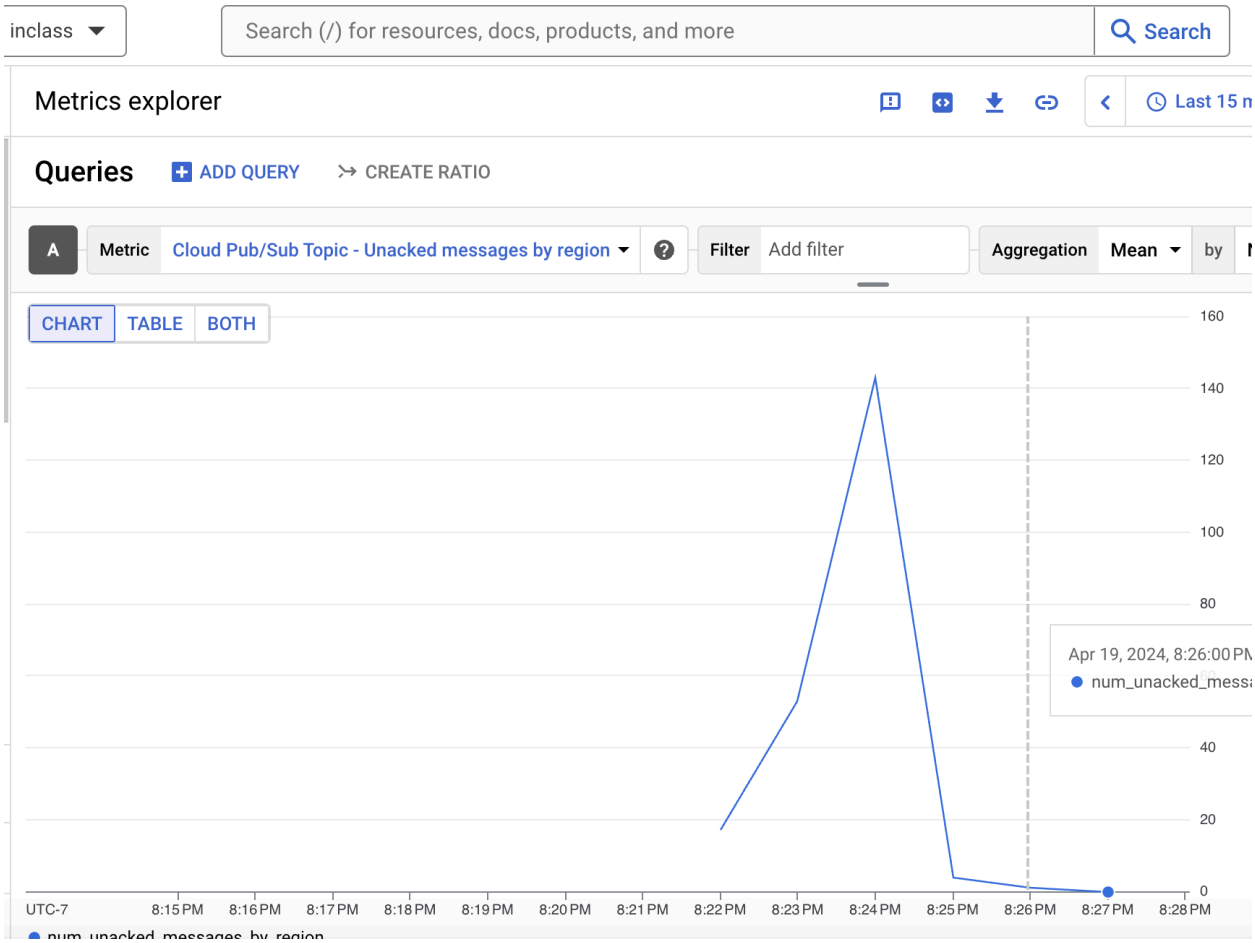


4. Create a “topic_clean.py” receiver program that reads and discards all records for a given topic. This type of program can be very useful for debugging your project code.

E. [SHOULD] Multiple Publishers

1. Clear all data from the topic (run your topic_clean.py program whenever you need to clear your topic)
2. Run two versions of your publisher concurrently, have each of them send all of your sample records. When finished, run your receiver once. Describe the

results.



The two publishers when running together outran the subscriber acknowledging them. It was observed that the unacked messages count kept increasing until the publisher stopped executing. After that point, the subscriber was able to steadily acknowledge all of the messages until the timeout.

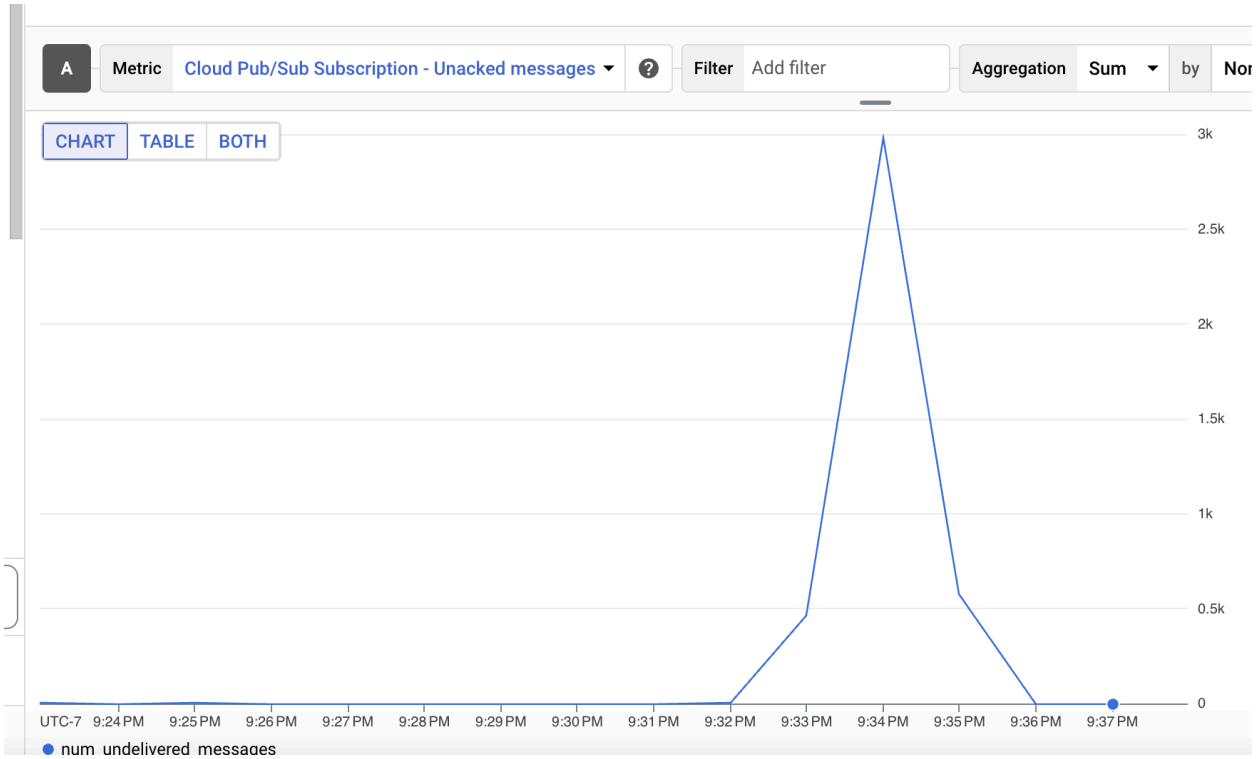
F. [SHOULD] Multiple Concurrent Publishers and Receivers

1. Clear all data from the topic
2. Update your publisher code to include a 250 msec sleep after each send of a message to the topic.
3. Run two or three concurrent publishers and two concurrent receivers all at the same time. Have your receivers redirect their output to separate files so that you can sort out the results more easily.
4. Describe the results.

I ran two publishers and two subscribers that output the list of messages they acknowledge to a text file. It can be observed that both the subscribers acknowledge

the messages at the same rate, and under a period of ~50 secs the following compares their output file's no. of lines (i.e. factor of no. of messages acknowledged).
(myenv) chetha@instance-20240419-022800:~$ wc -l output1.txt
10932 output1.txt
(myenv) chetha@instance-20240419-022800:~$ wc -l output2.txt
11070 output2.txt

The below chart shows how the two subscribers acknowledged the messages equally at the same rate.



F. [ASPIRE] Multiple Subscriptions

1. So far your receivers have all been competing with each other for data. Next, create a new subscription for each receiver so that each one receives a full copy of the data sent by the publisher. Parameterize your receiver so that you can specify a separate subscription for each receiver.
2. Rerun the multiple concurrent publishers/receivers test from the previous section. Assign each receiver to its own subscription.
3. Describe the results.

When multiple subscriptions were used, it was clear how they received the same messages simultaneously from the Topic. Instead of the two subscribers acknowledging

the same set of messages, now they were acknowledging two different sets of messages. At a 50sec mark, the no. of acknowledged messages were almost the same.

(myenv) chetha@instance-20240419-022800:~$ wc -l output1.txt
25851 output1.txt
myenv) chetha@instance-20240419-022800:~$ wc -l output2.txt
25748 output2.txt