DataEng S24: PubSub

Vaishnavi Srinath
986561736

[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?

   Answer:
   If we run the receiver multiple times while only running the publisher once, then each instance of the receiver independently listens for messages on the subscription. Since there is only one publisher sending the messages, the messages are consumed by the receivers in a non-deterministic manner. This would depend on several factors like network latency, message size, time of message sent, time required to process message etc.
   When we run receiver multiple times, each receiver would receive a subset of message. The messages would not be evenly distributed among receivers always. There are also chances that the messages be processed multiple times and at the same times some messages may not be processed at all if the receivers terminate before consuming all messages.

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

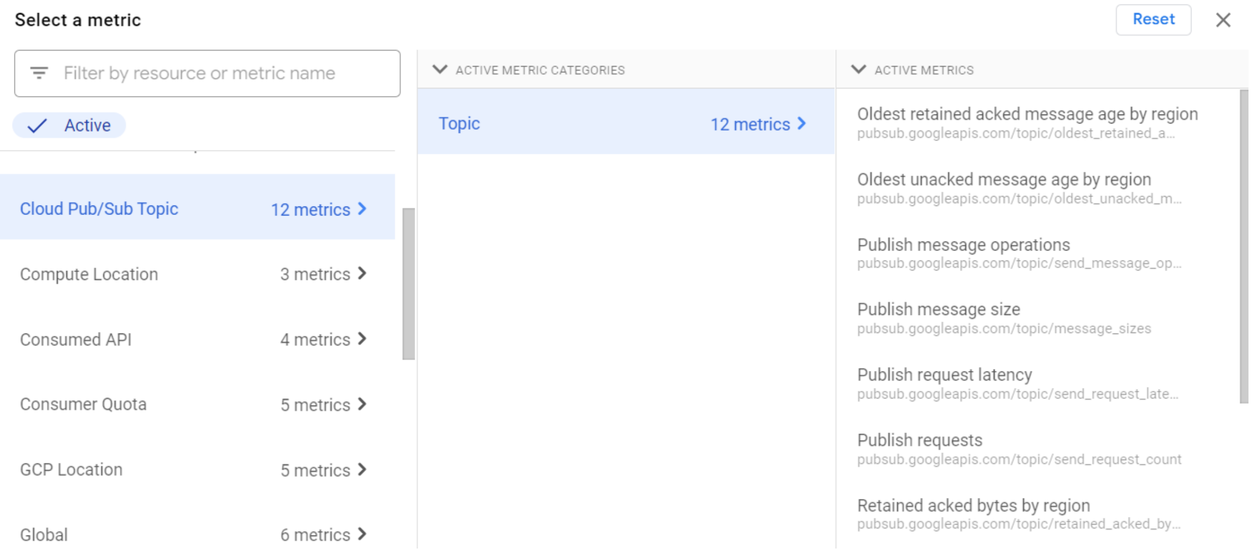
   Answer:

   - To ensure proper delivery, the data is written to storage. A publishing forwarder initially writes the messages to N clusters(where N is odd) until the message is written to atleast N/2 clusters. These clusters are present in data centers. Within each cluster, the message is written to M independent disks to ensure redundancy and fault tolerance. The data needs to be on M/2 disks before it is said to be existing in that cluster. So the message published will be written to atleast M/2 independent disks in

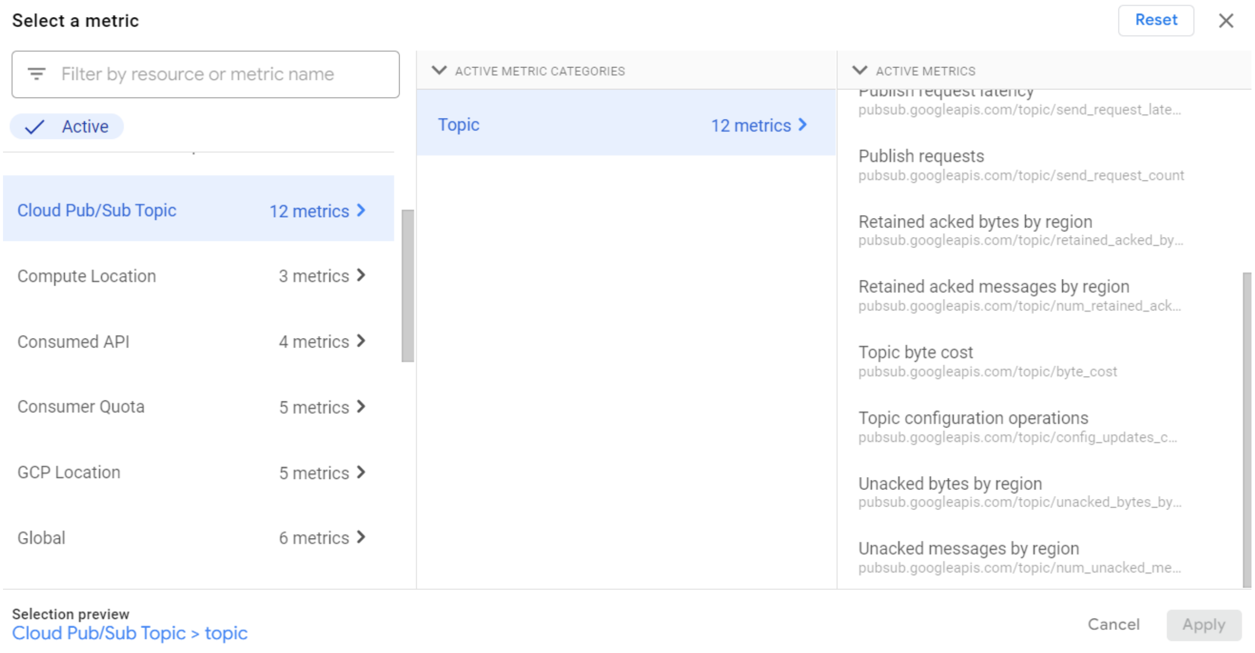
N/2 clusters before it is considered to exist on the storage. The data is then replicated to multiple disks (N*M disks in total) across clusters for durability.

- Also, there is a way to store messages even after it is consumed by the subscriber once to allow them to replay the messages once again. This can be done by configuring PubSub service with retention policy.

3. Is there a way to determine how much data PubSub is storing for your topic? Do the PubSub monitoring tools help with this?
   Answer:
   Yes, Google Cloud provides monitoring tools to determine how much data PubSub is storing. It provides a tool names 'Metrics'. This exposes various metrics which we can use to monitor the size of the backlog for a particular topic. The below screenshot shows the various kinds of monitoring tools provided. Thy can be used according to our needs.



Select a metric
Reset
×
Filter by resource or metric name
✓ Active
Cloud Pub/Sub Topic 12 metrics >
Compute Location 3 metrics >
Consumed API 4 metrics >
Consumer Quota 5 metrics >
GCP Location 5 metrics >
Global 6 metrics >
ACTIVE METRIC CATEGORIES
Topic 12 metrics >
ACTIVE METRICS
Oldest retained acked message age by region
pubsub.googleapis.com/topic/oldest_retained_a...
Oldest unacked message age by region
pubsub.googleapis.com/topic/oldest_unacked_m...
Publish message operations
pubsub.googleapis.com/topic/send_message_op...
Publish message size
pubsub.googleapis.com/topic/message_sizes
Publish request latency
pubsub.googleapis.com/topic/send_request_late...
Publish requests
pubsub.googleapis.com/topic/send_request_count
Retained acked bytes by region
pubsub.googleapis.com/topic/retained_acked_by...



Select a metric

Reset

Filter by resource or metric name

✓ Active

Cloud Pub/Sub Topic	12 metrics ›
Compute Location	3 metrics ›
Consumed API	4 metrics ›
Consumer Quota	5 metrics ›
GCP Location	5 metrics ›
Global	6 metrics ›

ACTIVE METRIC CATEGORIES

Topic	12 metrics ›

ACTIVE METRICS

Publish request latency
pubsub.googleapis.com/topic/send_request_late...

Publish requests
pubsub.googleapis.com/topic/send_request_count

Retained acked bytes by region
pubsub.googleapis.com/topic/retained_acked_by...

Retained acked messages by region
pubsub.googleapis.com/topic/num_retained_ack...

Topic byte cost
pubsub.googleapis.com/topic/byte_cost

Topic configuration operations
pubsub.googleapis.com/topic/config_updates_c...

Unacked bytes by region
pubsub.googleapis.com/topic/unacked_bytes_by...

Unacked messages by region
pubsub.googleapis.com/topic/num_unacked_me...

Selection preview
Cloud Pub/Sub Topic > topic

Cancel Apply

We can create custom dashboards too to monitor the PubSub using visuals and create alerts too. Another option is to by using the concept of logging. We can use logs to analyze the messages delivery and consumption patterns which can tell us how much data is being stored.

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.

Answer:

Two publishers are concurrently sending messages to the same topic, my-topic, with each publisher handling a separate set of records for vehicle IDs 3940 and 3257. The messages are being sent to the Pub/Sub service and written to storage for delivery to subscribers. When I run the receiver, it connects to the subscription associated with my-topic and starts consuming messages. Since each publisher is sending messages for a different set of vehicle IDs, the receiver receives a mix of messages for both vehicle IDs

3940 and 3257. Also, the unacked messages count kept increasing until the publisher stopped executing.

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.
   Note: I redirected the output using a command in VM. I haven't included that in the python file.
4. Describe the results.

Answer:

Messages from publishers are not delivered to receivers in the exact order they were published as multiple publishers are involved. This is because most systems buffer or queue messages before delivery, and the order of retrieval from the buffer might not be the insertion order. All the messages are acknowledged.

G. [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.

Answer:

For this, I created a new subscription id, my-sub1, and used the two subscription id's to run a separate subscription for each receiver.

```
python pub.py & python pub1.py & python sub.py my-sub > receiver_output1.txt &
python sub1.py my-sub1 > receiver_output2.txt &
```

I see that receivers process only the relevant messages. So the overall efficiency is increased and unnecessary processing is reduced. Here each receiver subscribes to a separate subscription id. So, the messages are sent to the relevant receiver only i.e there exists targeted delivery.