DataEng: Data Maintenance In-class Assignment

This week you will gain hands-on experience with Data Maintenance by constructing an automated data archiver that compresses, encrypts and stores pipelined data into a low-cost, high-capacity GCP Storage Bucket.

Submit: 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 any needed code before submitting for this week.

Your job is to develop a new, separate python Kafka consumer similar to the consumers that you have created multiple times for this class. This new consumer should be called archive.py and should receive all data from a test Kafka topic, compress the data, encrypt the data (optional) and store the compressed data in a GCP Storage Bucket.

Note that each member of your team should build their own archive mechanism. As always, it is OK to help one another, but each person should develop their own python program for archiving.

Discussion Question for Your Entire Group (do this first)

When archiving data for a data pipeline we could (a) compress, (b) encrypt and/or (c) reduce the data. Here, “reducing the data” refers to the process of transforming detailed data, such as 5 second breadcrumbs for all buses on all trips, into coarser data that contains only a subset of the original data such as only some buses, some trips or possibly fewer breadcrumbs per trip.

Under what circumstances might each of these transformations (compress, encrypt, reduce) be desirable for a data archival feature?

In data archiving, different transformations have different uses:

1. **Compression:** This is helpful when you have a lot of data and want to save space. It makes files smaller so they use less storage and can be moved around more quickly.
2. **Encryption:** This is useful when you're storing sensitive data and need to keep it secure. It scrambles the data so only someone with the right key can read it, keeping it safe from unauthorized access.
3. **Reduction:** This can mean removing duplicate data or summarizing it in some way. It's good when you have a lot of data and want to keep the important parts, but without taking up as much space.

Part 1: Data Archival to Cold Storage

A. Create test topic

Create new Kafka producer and consumer programs as you did with the Data Transport in-class activity. Create a new Kafka topic that is separate from the topic(s) used for your project. Call it “archivetest” or something similar. As with the Data Transport activity you should initially have your new producer produce test data and have a single consumer/archiver (call it “archiver.py” or something similar) that consumes any/all data sent to the Kafka topic and simply prints out the data consumed.

B. Create separate consumer groups

Create two separate Consumer Groups for your new Kafka topic. Run a separate consumer for each group and verify that each consumer consumes all of the data sent by the producer. The first of these two consumers will simulate your primary pipeline’s consumer. In the next step you will modify the second consumer to archive the data.

C. Archive the data

Modify the second of the two consumers (rename it “archiver.py” or something similar) to store all received data into a file.

Next, modify the archiver to store all received data to a GCP Storage Bucket. You will need to create and configure a Storage Bucket for this purpose. You are free to choose any of the available storage classes. We recommend using the Nearline Storage class for this assignment. Be sure to remove the bucket at the end of the week to reduce GCP credit usage.

```
import os
from google.cloud import storage



def upload_blob(bucket_name, source_file_name, destination_blob_name):
    """Uploads a file to the bucket."""
    storage_client = storage.Client()
    bucket = storage_client.bucket(bucket_name)
    blob = bucket.blob(destination_blob_name)
    blob.upload_from_filename(source_file_name)

    print(f"File {source_file_name} uploaded to {destination_blob_name}.")

def get_data_from_source():
    with open('source_data.txt', 'r') as f:
        data = f.read()
    return data

def main():
    # Get data from your source
    received_data = get_data_from_source()

    # Writing received data to a file
    with open('data.txt', 'w') as f:
        f.write(received_data)

    # Uploading the file to Google Cloud Storage
    upload_blob('archive-bucket', 'data.txt', 'data.txt')

# Call the main function
if __name__ == '__main__':
    main()
```

D. Compress (Optional)

Modify your archiver program to compress the data before it stores the data to the storage bucket. Use zlib compression which is provided by default by python. How large is the archived data compared to the original?

E. Encrypt (Optional)

Modify your archiver to encrypt the data prior to writing it to the Storage Bucket. Your archive.py program should encrypt after compressing the data. Use RSA encryption as described here: link There is no need to manage your private encryption keys securely for this assignment, and you may keep your private key in a file or within your python code.

Be sure to test your archiver by decrypting and decompressing the data stored in the Storage Bucket. We suggest that you create a separate python program for this purpose.

How large is the archived data?

F. Add Archiving to your class project (Optional)

Add an archiver to your class project's pipeline(s). Mention this in your final project presentation video. You should only need one archiver for the entire project, so coordinate with your teammates if you choose to take this step. For the class project, it is not necessary to securely manage your RSA private encryption key, and it is OK to keep it in a file or in your python source code.