Data Engineering II - 1TD076

Assignment 2 - A Practical Introduction to Apache Pulsar

Teame Hailu Gebru

April 24, 2025

Task 1: Setting up Apache pulsar

A. Start Pulsar in the bare VM

Pulsar has been started successfully in standalone mode. Part of screenshot showing output for successful setup is shown in figure 1.

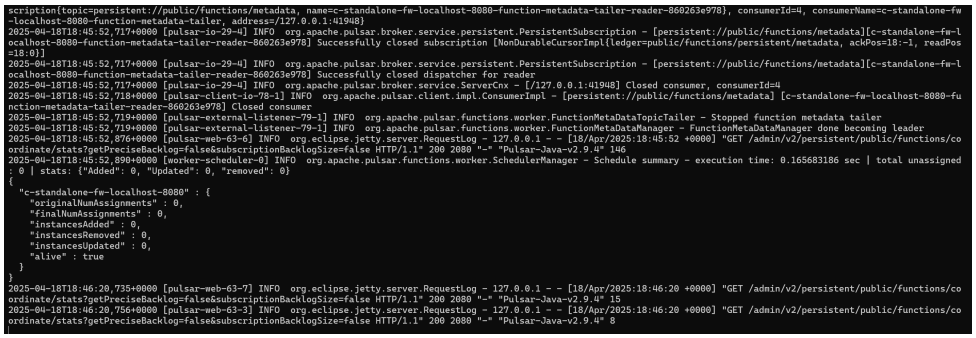

Figure 1: Output for successful setup of Apache pulsar

B. Setting up pulsar in a container in a VM

The setup of Apache Pulsar in a Docker container within a virtual machine was successfully started, as shown in figure 2.

Figure 2: Output for successful setup of Apache pulsar in Docker container

Task 2: Producing and Consuming messages

A. Running Apache pulsar, consumer and producer on a single machine

After creating consumer.py and producer.py files in two terminal in the same VM and successfully started as shown in the figure below.

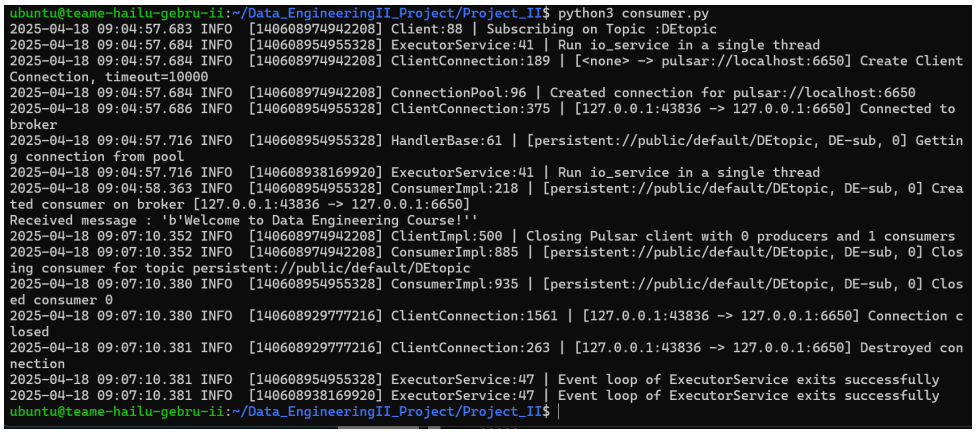

Figure 3: Output for successful setup of consumer

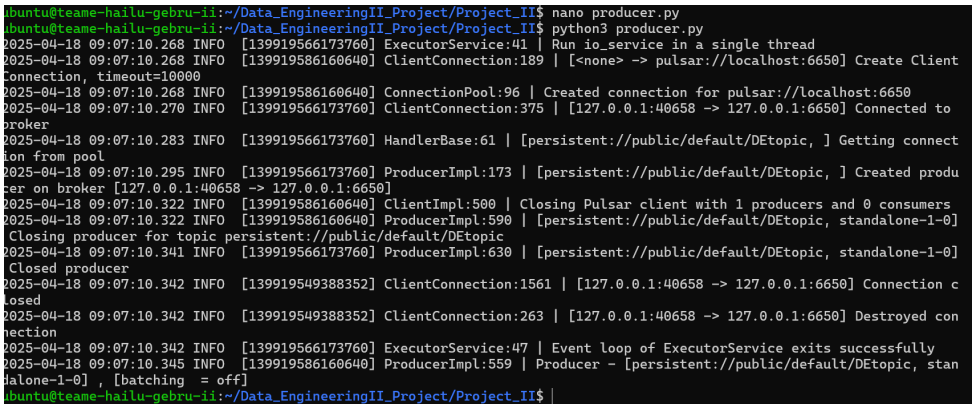

Figure 4: Output for successful setup producer

Task 3: Conceptual questions

1. What features does Apache Pulsar support have which the previous distributed data stream framework (e.g., Kafka) does not support?

Apache Pulsar support several features that enhance flexibility and scalability compared to the previous data frame works:

- **Multi-tenancy:** Unlike Kafka, Pulsar supports multi-tenancy, allowing multiple users to share the same Pulsar cluster securely.
- **Built-in geo-replication:** Pulsar ensures data availability across multiple geographical locations through its built-in support for geo-replication. This feature enables replication of data across data centers without the need for additional tools, making it highly effective when producers and consumers are distributed across different locations.
- **Segmented architecture (Broker + BookKeeper):** In contrast to Kafka's monolithic broker+storage approach, Pulsar separates serving (brokers) and storage (BookKeeper), which improves scalability and resilience.

2. What is the issue with using a batch processing approach on data at scale? How do modern data stream processing systems such as Apache pulsar can overcome the issue of batch processing?

Batch processing introduces latency and is inefficient for real-time analytics, especially as data volume grows. It waits for all data to arrive before processing, causing delays and potential system bottlenecks.Modern data processing systems overcome this by:

- trying to combine batch and stream processing capabilities into a single or multiple parallel data processing pipelines.
- Enabling real-time message streaming, where data is processed immediately upon arrival and meets the challenges of incremental processing, scalability, and fault tolerance.

- Supporting low-latency communication between producers and consumers, evident from the immediate message transmission in the consumer terminal output.
- Allowing fine-grained scaling, so system resources are used more efficiently over time, unlike static batch jobs.

By using streaming, Pulsar facilitates quicker insight generation and supports time-sensitive applications like fraud detection or live monitoring.

3. What is the underlying messaging pattern used by Apache Pulsar? What is the advantage of such a messaging pattern?

Apache Pulsar uses the publish-subscribe (pub-sub) messaging pattern. In this pattern, producers publish messages to logical channels called topics. Consumers can subscribe to these topics and receive messages in real-time.

The main advantage of the pub-sub pattern is decoupling—producers and consumers do not need to know about each other. This enables better scalability, parallel processing, and system flexibility.

4. What are different modes of subscription? When are each modes of subscription used?

- Exclusive: Only one consumer can subscribe to a topic per subscription. Ideal for guaranteed single processing.
- Failover: One active consumer, others standby. Ensures high availability.
- Shared (Round-robin): Messages are distributed across multiple consumers. Used for parallel processing.
- Key_Shared: Messages with the same key go to the same consumer. Best for ordered processing based on keys.

5. What is the role of the ZooKeeper? Is it possible to ensure reliability in streaming frameworks such as Pulsar or Kafka without ZooKeeper?

ZooKeeper is used for:

- Cluster coordination:It keeps track of which Pulsar brokers are active and manages leader election for components like the Pulsar broker service
- Metadata storage:Zookeeper stores important metadata such as topic ownership, namespace information, and broker assignments.
- configuration Management:It helps in managing configuration changes across the cluster consistently.

While Pulsar originally used ZooKeeper, newer versions aim to eliminate this dependency via the Pulsar broker-based metadata management. This makes deployment lighter and improves fault tolerance.

For small-scale deployments ZooKeeper may still be bundled or optional. For large, production-grade clusters, decoupling ZooKeeper improves maintainability.

6. Enlist different components of Pulsar?

1. Broker – Handles message dispatch between clients and BookKeeper.
2. BookKeeper – Manages persistent message storage.
3. ZooKeeper – Manages cluster metadata (optional in modern setups).
4. Topic – Logical channel for messages.
5. Namespace – A grouping mechanism within a tenant.
6. Tenant – Multi-tenant boundary (not explicitly used in Task 1 or 2).

7. Mention what is the meaning of broker in Pulsar?

A broker is a core component that acts as a message router between producers and consumers.The key responsibilities of pulsar broker are:

- Receives messages from producers.
- Dispatches messages to consumers.
- Handles subscription and delivery logic.

8. What is the role of a tenant in pulsar?

A tenant represents an isolated unit in a multi-tenant Pulsar deployment and supports isolation, authentication,authorization and quotas.

- Has its own namespaces, topics, and roles.
- Useful for multi-user environments.

9. What will happen if there is no log compaction in Apache Pulsar?

Pulsar retains all historical messages.

- Storage may fill up rapidly, increasing latency and cost.
- Consumers re-reading messages might encounter redundant or outdated data.

Log compaction helps maintain a compacted view of the latest key-value pairs—critical in data consistency for large-scale systems.

Task 4: Splitting and merging operations with Apache Pulsar

Question 1: Issue in Current Implementation

The current implementation of `conversion.py` processes each word of a string sequentially using a for-loop. This approach works for small-scale data but is not scalable for large datasets, especially when the number of words reaches millions.

The main issues include:

- **Lack of Parallelism:** The conversion operation is applied sequentially to each word. This significantly slows down processing for large input sizes.
- **Memory Constraints:** The entire input and output data must be stored in memory, which is inefficient for large-scale text data.
- **No Distribution:** The implementation is limited to a single machine, making it unable to leverage distributed computing environments.
- **No Fault Tolerance:** If the process fails midway, all data must be reprocessed, as there is no message-level reliability.

These limitations make the current approach unsuitable for handling big data in real-time or large-scale scenarios.

Question 2: Redesign using Apache Pulsar

To address the scalability and performance limitations in the current implementation, Apache Pulsar can be used to split and merge data operations across multiple components.

a. Splitting and Merging using Pulsar

The system is redesigned as follows:

- **Producer:** Splits the input string into individual words and sends each word as a separate message to a Pulsar topic (`conversion-topic`).
- **Consumers:** Multiple consumers subscribe to the topic and apply a conversion operation (e.g. uppercase) on each word.
- **Merger:** A final component (or consumer) subscribes to the processed word topic (`result-topic`) and reconstructs the full converted string.

This architecture enables parallel processing and efficient message handling.

b. Architecture Diagram

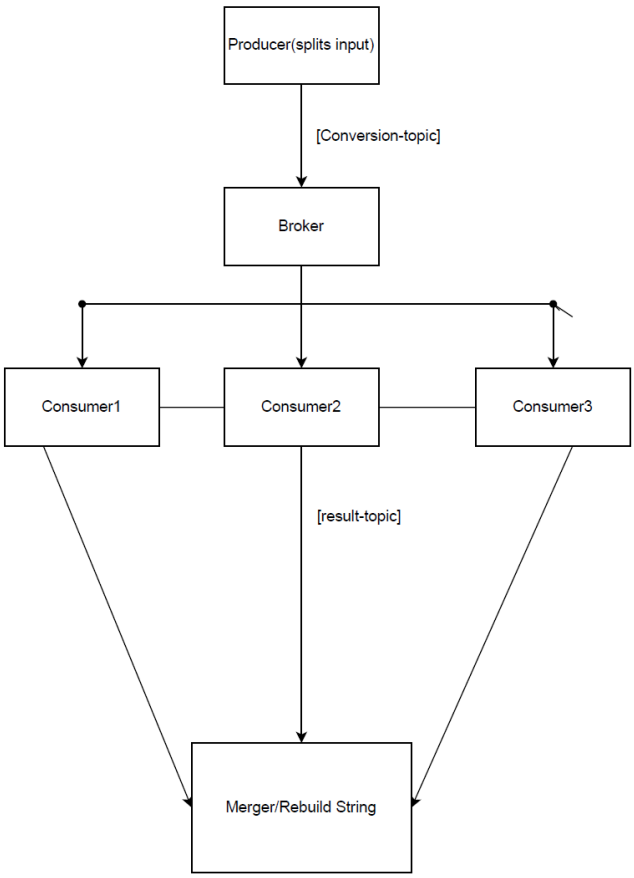

Figure 5: Architecture diagram

c. Labels

- **Broker:** Apache Pulsar broker that routes messages between producers and consumers.
- **Producer:** Sends individual words to the topic.
- **Consumers:** Perform conversion logic and send results to another topic.
- **Merger:** Reassembles the final output string from converted messages.

This redesigned system provides scalability, parallelism, and better handling of large-scale data streams.

1 Question 3. Implementation Overview (Python)

All components were implemented using the Apache Pulsar Python client (pulsar-client==2.9.4) on a single VM using multiple terminal sessions.

producer.py: Splits the string and sends each word to conversion-topic.For python script refer appendix A.1.Output showing successful execution of producer script is shown in figure 6 below.

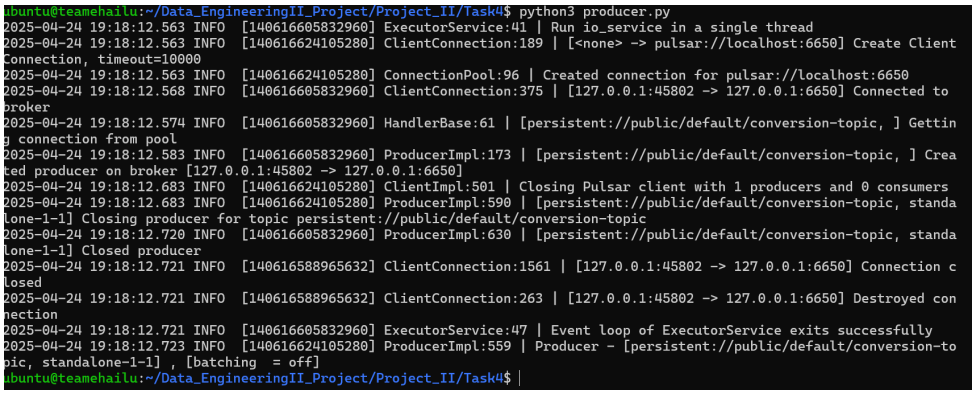

Figure 6: Output showing the successful execution of the producer script in the Pulsar-based system.

conversion_consumer.py: Listens to conversion-topic, converts each word to uppercase, and sends to result-topic.For python script refer appendix A.2.Output showing successful execution of consumer script is shown in figure 7 below.

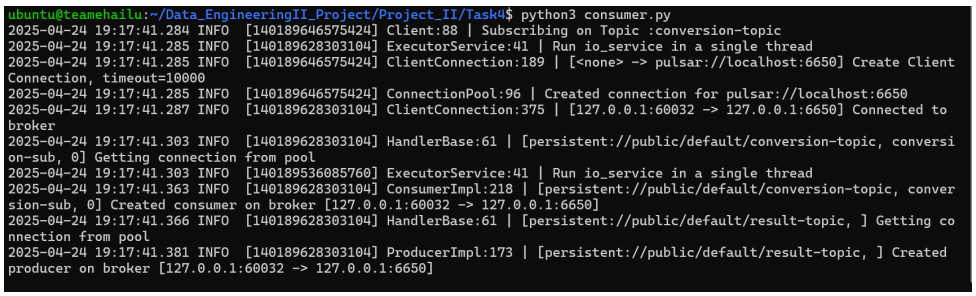

Figure 7: Output showing the successful execution of the consumer script in the Pulsar-based system.

merger.py: Listens to result-topic and reconstructs the final sentence.For python script refer appendix A.3.Output showing successful execution of merger script is shown in figure 8 below.

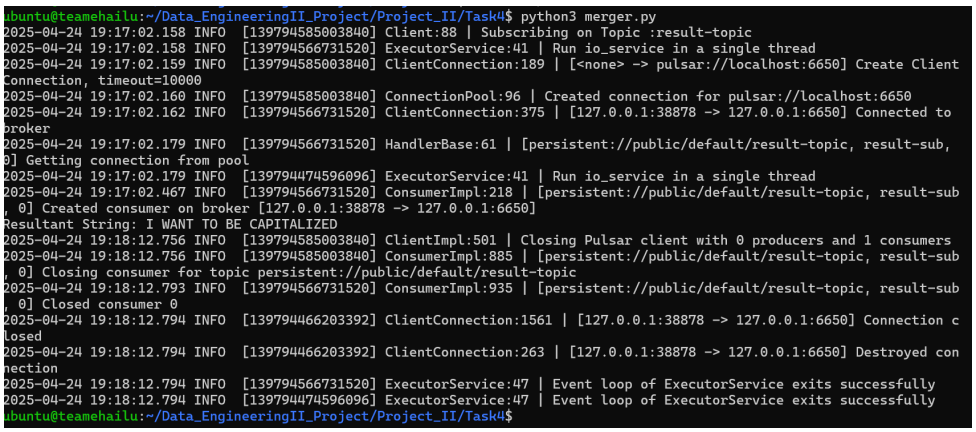

Figure 8: Output showing the successful execution of the merger script in the Pulsar-based system.

Each conversion consumer runs independently, and multiple instances can be launched in parallel

to support horizontal scaling using Pulsar's Shared subscription mode.

A Appendix

A.1 Producer

```
import pulsar

INPUT_STRING = "I want to be capitalized"
client = pulsar.Client('pulsar://localhost:6650')
producer = client.create_producer('conversion-topic')

for word in INPUT_STRING.split():
    producer.send(word.encode('utf-8'))

client.close()
```

A.2 Conversion_consumer

```
import pulsar

def convert(word):
    return word.upper()

client = pulsar.Client('pulsar://localhost:6650')
consumer = client.subscribe('conversion-topic', subscription_name='conversion-sub')
producer = client.create_producer('result-topic')

while True:
    msg = consumer.receive()
    try:
        converted = convert(msg.data().decode('utf-8'))
        producer.send(converted.encode('utf-8'))
        consumer.acknowledge(msg)
    except:
        consumer.negative_acknowledge(msg)

client.close()
```

A.3 Merger

```
import pulsar

client = pulsar.Client('pulsar://localhost:6650')
consumer = client.subscribe('result-topic', subscription_name='result-sub')

result = []
# For demo: assuming we know the number of words
expected_words = 5

for _ in range(expected_words):
```

```
    msg = consumer.receive()
    try:
        result.append(msg.data().decode('utf-8'))
        consumer.acknowledge(msg)
    except:
        consumer.negative_acknowledge(msg)

print("Resultant String:", ' '.join(result))
client.close()
```